PROGRAMMABLE CYBER NETWORKS FOR CRITICAL INFRASTRUCTURE

BY

RAKESH KUMAR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

ABSTRACT

The operational integrity of the infrastructure systems is critical for a nation state's economic and security interests. Such systems rely on automation to perform complex control tasks to avoid malfunction caused by human error. However, such automation requires coordination of their components. Such coordination relies on several guarantees of safety and performance from their underlying cyber networks. It is challenging to provide such guarantees using traditional networks due to their rigid feature set and distributed, opaque and non-standard control interfaces. The central goal of this dissertation is to develop a set of design tools that use network programmability to achieve end-to-end delay, access control and resiliency guarantees for critical infrastructure (CI) applications. We propose and evaluate the architecture and design of several tools to address these guarantees singularly and simultaneously.

With the standardized control and data-planes, the computational *analysis* of the centralized network configurations has emerged as a powerful approach for solving a variety of problems. We used this approach in one of our analysis tools to simultaneously validate access control and resilience of networks. We also used an analytic approach to assess the resiliency of CI network with the use of metrics computed by using Monte Carlo methods. To that end, we built a data-plane simulator to enable such computation.

Furthermore, it is now feasible to *synthesize* desired behavior in a programmable CI network to meet the performance and resilience goals for individual application flows. We used the synthesis approach to build a tool that uses efficient centralized algorithms to synthesize control-plane configuration resulting in flows meeting their end-to-end delay deadlines. We also demonstrate a framework that uses synthesis at the intersection of control and data planes to implement network coding to achieve seamless resiliency for network flows.

To my family: Komi, Mummy, Papa and Sandhi
and
To my teachers

ACKNOWLEDGMENTS

I would like to thank my advisor for his support and guidance throughout the research and writing of what has become this dissertation. I would like to thank my committee members for providing valuable insights and prodding me in the right direction. I would like to thank Dr. Rakesh Bobba, Dr. Sibin Mohan and Vignesh Babu for their collaboration and ideas. I would like to thank Prof. Agha, Prof. Srikant and Prof. Smaragdis for their generosity with their time.

I am immensely grateful for the people I met during the time I performed this research. My friends at work (Ashish, Elizabeth, Hoang Hai, Kartik, Tonia and Vignesh) provided me a community. I would like to thank my friends (Anna, Azeem, Faraz, Javier, Miriam, Om, Rajesh, Sarah and Tanya) for their company and persistent support.

Finally, I would like to thank the Department of Energy (DoE), Department of Homeland Security (DHS) and National Science Foundation (NSF) for funding the research grants that enabled this part of my journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
CI	Critical Infrastructure
RRP	Resilient Routing Policy
SDN	Software Defined Networking
NC	Network Coding
DC	Diversity Coding
COTS	Commercial Off-The-Shelf
FEC	Forward Error Correction
QoS	Quality of Service
NERC	North American Electrical Reliability Corporation
ESP	Electronic Security Perimeter
TCP	Transmission Control Protocol
IP	Internet Protocol
HTTP	Hyper-Text Transfer Protocol
PSA	Portable Switch Architecture

CHAPTER 1

INTRODUCTION

1.1 Motivations

Modern life is anchored in the seamless operation of infrastructure systems. The operational integrity of such systems is at the heart of a nation state's economic and/or security interests [1] and consequently they are designated as critical infrastructure (CI) [2] [3]. CI systems are classified into various sectors and range from those directly interacting with the daily life of their end-users (e.g. power grid, wastewater treatment plants, financial networks) to sectors that enable other sectors in the wider economy (e.g. advanced manufacturing or emergency services). These systems increasingly rely on computerized automation to perform the critical control tasks because it makes them less prone to faults caused by human mistakes.

Hence, such automation systems inevitably require using a computer network to coordinate their various components and therein lies the biggest threat to them. For example, the modern power grid relies on sensing and control units communicating over a cyber network to derive *smart* outcomes for the utilities, generators and consumers and avoid blackouts [4]. Similarly, advanced industrial control systems use cyber networks to orchestrate physical units (e.g. robots, assembly lines) to increase productivity and reduce losses. These networks have the following distinct requirements:

- **Security**: CI systems are constantly under attack [5] [6]. Some of these threats are mitigated by imposing access-control using firewalls. Some CI networks (e.g. power grid networks, hospital computer networks) undergo routine audits [7] [8] of their firewall configuration to determine if the critical assets are adequately protected. However, continued successful attacks have demonstrated the inadequacy of audit processes to secure these networks. Furthermore, due to the possibility of human errors in the design process [9], automated tools are required to validate network device configurations to ensure that the access control on such networks is correctly implemented.

- **Resilience**: CI systems require that their networks continue to operate seamlessly even when a subset of network devices or links fail. When such failures occur, the routing of packets needs to adapt so as to avoid using the failed components of the network while using the least amount of redundancy. However, such efficient adaptation must occur at time-scales that necessitate making decisions on locally on the network device [10].

- **Bandwidth and End-to-End Delay**: CI systems need guaranteed bandwidth and upper-bound on end-to-end delays for network flows [10]. Historically this has been achieved by overprovisioning of network and use of specialized hardware (e.g., AFDX in avionics [11]). More recently, standards bodies have also tried to develop standards for such special-purpose hardware [12] [13]. However, overprovisioning the resources and the use of specialized hardware can be prohibitively expensive. Hence, design tools are required that can use commodity hardware and perform resource allocation on that hardware such that the network meets its performance guarantees.

In order to meet the requirements described above simultaneously, the networks have to grow more feature-rich. More features require a more precise control of the behavior of individual network devices (e.g. firewall, switches etc.) that constitute the network. However, traditional networking devices have a rigid feature set. The APIs to control their features are either non-existent or non-standardized. Consequently, it has been hard to evolve the capabilities of the networks without operational disruptions. Historically, it has required clever mechanisms on the part of researchers to overcome these obstacles [14]. This may finally be changing with the advent of programmable networks [15] [16].

1.2 Programmable Networks & Critical Infrastructure

There are various interpretations of what programmable networks are. However, for the purposes of this dissertation, this term refers to the networks where intent of the network designer or operator is *programmed* onto the network using a collection of application programming interfaces (APIs). Such networks comprise of two distinct architectural planes, each of which can be modified using the API it exposes. The control-plane decides *what* functions each network device (called *switch*) performs whereas the data-plane determines *how* the function is performed at each switch. Naturally, the data-plane is located at the switches to maintain speed for making per-packet decisions whereas a logically centralized controller houses the control-plane. The controller and the switch are able to communicate with each other securely using a secure API. Both switches and controllers are typically built on top of commodity off-the-shelf (COTS) hardware.

In the last decade, network programmability has permeated in the design and operation of various types of networks. This fundamental shift away from the distributed and proprietary control-plane architecture is driven by a combination of open software and standardization efforts. Open-source control-plane has been developed and adopted by consortia of users, academia and equipment vendors to promote software reuse and robustness [17] [18]. Standards such as OpenFlow [19] and P4 [20] have enabled an unprecedented data-plane flexibility. This combination has allowed proliferation of new features in production networks in data centers [21], wide-area networks [22], and the internet exchange points [23].

Similarly, programmability has profound implications for the CI networks. It allows exploration of new design trade-offs to meet their unique requirements. However, despite there being many potentially applicable use-cases, the networks in CI have been slow to adopt programmability. One example of such a use case is the NERC access-control regulations [7] for the critical assets in the Electronic Security Perimeter (ESP). The regulation requires limited and well-documented access to these hosts from the power utility's corporate network. However, despite there being financial penalties when a violation is detected, several utilities have been found to violate this regulation [24] as recently as in 2018. While the problem of enforcing an access-control policy in real-time has been solved with the use of programmable control-plane [16] [25], the CI network operators need regulatory compliance for control-plane output during the design phase *before* the network is put in production.

The design emphasis of CI networks must not be on simply using new features and mechanisms to solve such use-cases. Historically, addition of new features and mechanisms has caused networks to become more complex [26]. One way to tame this complexity is for networks to become more programmable because it provides operational flexibility. However, such flexibility does will not automatically *guarantee* that a CI network will meet its intended requirements. This dissertation is an attempt at bridging the gap between the flexibility made available by programmability and the guarantees required by the CI networks.

1.3 Dissertation Objective

In order to provide guarantees in CI networks, we focus on the design phase prior to the implementation of a network. This is also when capacity planning, critical asset designations etc. are performed. The central goal of this dissertation is to develop a set of design tools that guarantee a CI network shall meet its access-control, end-to-end delay and resilience requirements.

There are two distinct approaches that our tools take to exploit network programmability. One is the analytic approach, whereby a snapshot of the logically centralized state of the network is analyzed to validate that it meets certain requirements. Alternatively, there is the synthetic approach, whereby the network state is synthesized to meet its requirements in a way which is correct by design. The next sub-section summarizes our contributions.

1.4 Summary of Contributions

As stated previously, our contributions are a series of design tools that target a set of requirements that are unique to CI networks. Each requirement concerns a different behavior of the network and thus requires its own model. For example, for access-control, the packet transformations that occur within a switch are crucial, whereas they may not have direct implications for the data rate at the output of the switch. Hence, we developed each design tool to encapsulate an appropriate model to represent the network for the particular requirement it addresses. Besides the models, these tools also use the appropriate algorithms to explore various design trade-offs. These trade-offs are not novel, however, the rise of network programmability has made it possible to explore them very cost-effectively. For example, if a packet flow is to meet its end-to-end delay deadline, there is a fundamental trade-off between the flow rates and total available network flow. Our models and algorithms formalize these trade-offs. Finally, we evaluate the tools using simulations and empirical experiments. Our contributions are summarized in Table 1.1.

Table 1.1: Summary of contributions: The numbers in the cells refer to the list below. The shaded cells represent existence of previous work that has addressed the problem.

Guarantee	**Analysis**	**Synthesis**
Access Control	(1)(3)	▨
Failure Resiliency	(1)(2)(3)(6)	(4)
Bandwidth and Delay Guarantee	▨	(5)

Following is the list of publications from this work:

1. **Rakesh Kumar**, David M. Nicol: *Validating Security and Resiliency in Software Defined Networks for Smart Grids*, IEEE Transactions on Smart Grid (in review)

2. David M. Nicol and **Rakesh Kumar**: *Efficient Monte Carlo Evaluation of SDN Resiliency*, Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation. ACM, 2016.

3. **Rakesh Kumar** and David M. Nicol: *Validating Resiliency in Software Defined Networks for Smart Grids*, IEEE International Conference on Smart Grid Communications, 2016.

4. **Rakesh Kumar**, Monowar Hasan, Smruti Padhy, Konstantin Evchenko, Lavanaya Piramanayagam, Sibin Mohan, Rakesh B. Bobba: *End-to-End Network Delay Guarantees for Real-Time Systems using SDN*, IEEE Real-Time Systems Symposium, 2017.

5. **Rakesh Kumar**, Vignesh Babu, David M. Nicol: *Network Coding for Critical Infrastructure Networks (Extended Abstract)*, IEEE International Conference on Network Protocols, 2018.

6. David M. Nicol, **Rakesh Kumar**: *SDN Resiliency to Controller Failure in Mobile Contexts*, Winter Simulation Conference, 2019.

1.4.1 Validating Access Control and Resiliency

CI networks are required to provide connectivity that is resilient to link and device failures while simultaneously complying with an access control policy. The control-plane is responsible for implementing these requirements by way of generating the network's forwarding plane state. However, these requirements can be in conflict with each other because the resiliency might require using paths that are not desirable due to the access control policy. Thus, the flow rules synthesized by the control-plane may inadvertently choose one requirement over the other, thus leading to catastrophic failures and/or security breaches.

Our first contribution is a policy specification language that allows a user to specify their requirements. The specification allows expression of various types of requirements simultaneously. These requirements include access-control, resiliency from link and device failures, path lengths and link exclusion for certain flows. In order to ensure that the flow rules meet their network's requirements, they must be validated against a given policy. This dissertation contributes two distinct methods of performing such validation.

We perform two types of validation using distinct models motivated by differing set of requirements. In the first method, we perform exhaustive validation under all possible packets. However, each flow rule can address an arbitrary set of packets and has a priority and a set of actions associated with it. Hence, the number of ways flow rules can be generated is very large. This results in a combinatorial explosion in the number of possibilities that need to be validated. In order to tame this resulting complexity, we used header-spaces to represent a collection of packets and a directed graph to represent the flow rules and ports of the switches. While using this methods results in exhaustive validation, it can be computationally expensive. Hence, we used another method where we model individual packets and simulate the behavior of switches. While both of these methods are applicable for general networks, we use OpenFlow as the reference standard.

Finally, beyond statically validating a specific policy, it is important to quantify and characterize the nature of resiliency in a CI network using metrics. Again, due to the combinatorial explosion, these metrics are difficult to measure analytically. Hence, this distribution contributes methods to measure them empirically against a network's configuration using Monte Carlo method. We contributed several mechanisms of doing this using both simulation and exhaustive modeling described above.

1.4.2 Synthesizing for Bandwidth and Delay Guarantees

Applications in the CI networks need guarantees on the end-to-end delay for packets in a flow. The flows in the CI networks frequently have a predictable maximum traffic rate. The rate for an individual flow can be used to estimate its worse-case network bandwidth requirement at each link in its path from source to the destination. A flow's end-to-end delay requirement will be met if there is more bandwidth available at each link than what it needs and if the total path length is small enough so that the cumulative per-hop processing delay is less than its requirement.

Hence, providing a delay guarantee for each flow relies on the conservative use of the available network bandwidth so that its end-to-end delay requirements are met even if every other flow in the system produces traffic at the maximum rate for an extended period of time. Fundamentally, a network designer needs to decide what paths a given set of flows must take and then make the appropriate bandwidth allocations along these paths. Such granular allocation of bandwidth to an individual flow needs an open interface to each switch. The distributed control-plane does not provide such control. Hence, traditionally, the most widely used mechanisms either use redundant resources and/or specialized hardware.

However the programmable networks provide mechanisms to control the bandwidth, and consequently the delay, for individual traffic flows. Our approach to providing guarantees for end-to-end delay relies on such mechanisms. We solve a network-wide combinatorial optimization problem for allocating resources for each flow at each switch port. The constraints for each flow are its required bandwidth and delay. We use an NP-hard multi-constraint problem (MCP) formulation which can be solved with a greedy heuristic. Once the problem is solved, we enforce the solution by using the switch API. We synthesize the flow rules and use a separate queues for each individual flow to avoid interference between them. We also synthesize the configuration for mapping the flow traffic to the appropriate queue and bandwidth parameters for each queue at each switch.

1.4.3 Synthesizing for Resilience Guarantees

As noted above, applications in the CI networks need forwarding resilience to failure events that cause the network topology to change due to link or device failures. Such resilience implies that the packets in a given set of flows continue to be delivered while the event occurs. When a distributed control-plane is used, such an event results in routing table updates, and consequently the packets that are in-transit are lost. Similarly, even with the centralized control-plane and the use of fast-failover mechanism, there are packet losses due to time taken by the switch to shift traffic from one link to another. In either case, the effective forwarding state generated by the control-plane needs to be updated in the event of a failure, resulting in additional complexity in the control-plane. In the distributed control-plane, this complexity is handled by distributed algorithms that share link-state, while in the centralized control-plane, it is handled in a program that considers all possible combinations of failure events.

Furthermore, some applications may require the resilience to be *seamless*, so that not only does the network recover from a failure event, but there is no interruption in packet delivery as the network goes through such a topological change. Most modern networking is based on the store-and-forward paradigm so that the contents of a packet are either stored or forwarded by a network device such as switch, router, firewall etc. However, the store-and-forward paradigm cannot guarantee completely seamless packet delivery regardless of the location of the control-plane unless a complete duplication of network packets along multiple paths is used. This obviously leads to very inefficient use of network resources.

Hence to guarantee seamless resiliency, in contrast to the prevalent store-and-forward paradigm, we explored the use of algebraic coding across one or more packets at the switches. This approach has been previously explored in theory [27] [28], but its application has been limited to the application layer in part due to the complexity and rigidity of ASIC design. However, with the advent of programmable data-planes, it is now possible to use linear network coding (NC) at the intermediate network nodes to meet resilience requirements of the CI applications. To that end, we propose an architecture that realizes linear NC in programmable networks by decomposing the linear NC functions into the atomic coding primitives. We designed and implemented the primitives using the features offered by the P4 ecosystem. In our approach, a user modifies the data-plane and then uses the modified data-plane in the control-plane. With this combination, we demonstrate that we can synthesize flow rules that result in seamless resiliency when device/link failures occur.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows:

Chapter 2 introduces some preliminaries. Chapters 3 and 4 present the use of the analytic approach: Chapter 3 discusses validation of access-control and resilience guarantees in the configuration of CI programmable networks. Chapter 4 presents our work that enables the use of Monte Carlo methods to measure resiliency metrics.

Chapters 5 and 6 take a synthesis approach. Chapter 5 discusses our work to synthesize a configuration that meets bandwidth and delay guarantees. Chapter 6 describes our work to synthesize resilience guarantees using network coding. Chapter 7 concludes and proposes directions of future work.

CHAPTER 2

BACKGROUND

In this chapter, we provide a brief overview of the unique networking needs of the applications that constitute the critical infrastructure systems. We provide details about how the operational integrity of a CI system depends on the continuous operation of its underlying communication network. Finally, we introduce some background on programmable networks.

2.1 Critical Infrastructure Applications

Over the past few decades, digital technology has permeated the CI systems. The most common example is the rise of supervisory control and data acquisition (SCADA) systems in the power grid [29], oil refineries [30] and water treatment plants [31]. There are multiple reasons for these trends motivated by the needs of the specific system. However, a common thread is the use of computers to perform programmable automation of the control mechanisms to regulate system safety and performance while achieving economies of scale.

Most CI systems are implemented as distributed cyber-physical systems (CPS). They are often composed of multiple distributed applications that coordinate their activity. Each distributed application controls an aspect of the system operations. For example, in a power grid sub-station, there is an application that controls the voltage on distribution lines. However, the grid itself has many sub-stations and there are control applications that perform load-balancing across multiple sub-stations and thus need to coordinate with the sub-station level control applications. Each of these central and sub-station level applications is divided in distributed software components that perform functions such as collecting sensor measurements (sensors), running the control algorithms (controller) and actuating on the control decisions (actuators). However, such coordination, both within an application and among different applications, requires a robust communication network.

The rise of automation in the CI systems has coincided with an increased use of standardized networking technology (e.g. TCP/IP). Both the applications and the communication network infrastructure have moved towards standardization. CI applications use specially designed communication protocols. Often these protocols cover functionalities that span both transport (L4) and network(L3) layers. Examples of such protocols include Modbus [32], DNP3 [33] and IEC 61850 [34]. These protocols use both unicast and multicast network flows for the control tasks. The protocols are interactive. The most common pattern of messages involves a central control node collecting some state from a set of sensors and dispatching commands to actuators. The communication can be triggered periodically or is event-driven depending on the application.

However, due to the criticality and time-sensitivity of such tasks, the safe operation of these protocols requires specific performance and security guarantees from their communication networks. This need for guarantees has led to proliferation of specialized hardware and software. While the specification of such hardware and software varies depending on the objectives of the given CI application, there are three broad classes of requirements which are common in a wide variety of CI applications. We present a brief overview of these common requirements below.

2.1.1 End-to-End Delay and Bandwidth

For CI applications, it is crucial to be able to receive sensor readings and dispatch commands to actuators within a very specific window of time. The exact deadline for safe operation may vary depending on the application, but the application does not have tolerance for additional delay as some other applications such as video streaming might have. The optimal operation of the application may be affected when its end-to-end delay requirement is not met. For example, one of the causes of the 2003 Northeast blackout was found to be lack of reliable real-time data for the control systems [4]. Hence, due to the very nature of their objective, the CI applications have a key requirement of the communication network, which is to provide guaranteed maximum end-to-end delay for a given bandwidth need.

In traditional networks, these requirements are codified in various standards for different CI applications. In some instances, power grid applications can require a worst-case end-to-end latency of 5 ms [10]. In traditional networks, these requirements have been met by using specialized hardware and/or conservative overprovisioning of network resources. While such conservative approaches are operationally expensive, they are also very opaque. Such opacity makes it hard to understand the impact of a change in the network on the CI system as a whole. Thus, it is not uncommon for network designers to deploy a new layer of networking equipment for each new application deployment while keeping the old and redundant gear operational as well.

2.1.2 Access Control

Due to their criticality, the communication networks in CI systems are targets for nation-state level adversaries [5] [6]. Such attacks have been shown to be effective in disrupting the operations in a variety of recent events. Network Access Control is the first line of defense against such attacks. Many CI systems are required by regulatory bodies to have specific network access control policies in place. For example, in the United States, the NERC-CIP [7] standard defines a logical Electronic Security Perimeter (ESP) in the heart of any power grid system and regulates access to the components in the ESP. The violators of the standard are subject to fines [24].

In traditional networks, the operators are required to understand various configuration languages and manually configure access control policies on individual network devices such as firewalls and routers. This process is widely known to be error prone [9].

2.1.3 Failure Resiliency

Like any other application, CI applications are disrupted when cyber network component failures occur. However, some CI applications are uniquely prone to cascading failures due to cyber-physical co-dependencies [35] [36]. For instance, a cyber network component in a power grid may be powered using the grid itself. Hence, when a cyber component goes down, it may cause failures of other cyber components. Hence, it is common practice for the communication network for such CI systems to be designed with the necessary redundancy that anticipates various conditions.

In traditional networks, resiliency is provided by duplicating networks in the most conservative case. Such redundancy, while effective, can incur large operational and infrastructure overheads, especially if the network is not built using commodity off-the-shelf (COTS) components. Furthermore, if such redundancy is not used, the network relies on the convergence of a distributed routing algorithm to resolve new paths after link or device failures. In general, in internet-wide networks, such convergence is known to take arbitrarily long time with no guarantees [37] [38].

2.2 Programmable Networks

Due to the interdependencies among the nodes in any computer network, a total overhaul of the architecture of any network poses a great challenge. Thus, transition of networks to become more programmable has been evolutionary. This transition has been motivated by use-cases whose impact was either limited in network sizes or represented minor changes to the network architecture or both. The notion of reprogramming a computer network dates to at least 1996. Motivated by the need for rapid innovation in the network infrastructure, in their seminal work on Active Networking (AN), Tennenhouse and Wetherhall proposed that packets themselves carry programs (called *capsules*) to modify the behavior of the intermediate network devices [39]. By 2005, the first attempts to separate the control and data planes were successfully demonstrated using systems known as SoftRouter [40] and RCP [41]. While there are many reasons why these efforts did not result in widespread adoption, one is a general lack of applications that may need these advances. Only in the last decade has scalable data plane performance been matched with appropriate standardization of the abstractions in APIs.

The modern programmable networks consist of two disaggregated architectural planes that have separate concerns [16] [42]: The control-plane decides which packet forwarding and transformations occur at the switches, while the data-plane is responsible for actual implementation of such actions. These planes are delineated such that the control-plane is centralized and is driven by a global network state [40] [41] [43]. The data-plane is located on individual switches and relies upon the control-plane for instructions to perform per-packet operations. These two architectural planes communicate using a *southbound* API. The control-plane applications communicate directly with the control-plane using a *northbound* API. Following are more details about each of these planes and their interfaces.

2.2.1 Control Plane

The control-plane decides *what* happens to packets when they arrive inside the network. Such decisions include packet paths, packet scheduling decisions, access control policy implementation etc. The control-plane itself is further subdivided into a controller and a series of stand-alone applications that implement specific functionalities.

Such division of labor allows independent evolution of various network functions and reduces complexity. For example, in a distributed control-plane, a variety of network functions require keeping track of the current network topology and thus they end up duplicating the efforts. However, in a centralized and programmable control-plane, the network state (including the topology) is logically centralized and available to all the applications. Thus, network functions which are implemented as control-plane applications only need to focus on their core application logic.

2.2.2 Data Plane

The data-plane implements *how* the actions decided by the control plane are applied to the packets when they arrive at an individual network node (i.e. switch). The design of a data-plane needs to maintain what is known as the line speed performance: that is, each packet is processed before the next one arrives, even if the device is operated at its maximum capacity. The programmable data-planes come in two varieties: they are either fixed-function such as OpenFlow [19] or they are programmable using standard interfaces such as the P4 language [20] [44]. The pertinent details of each

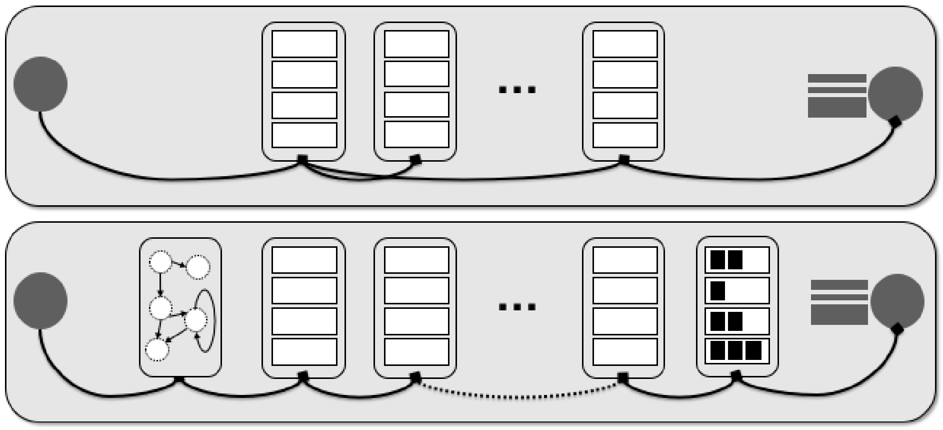


Figure 2.1: Data planes: standardized (top) vs. programmable (bottom).

kind of data-plane are as follows:

Standardized Data Plane

A switch with a standardized data-plane contains a single table processing pipeline and multiple physical ports as shown in top half of Figure 2.1. Packets arrive at one of the ports, and are processed by the pipeline comprised of one or more flow tables. Each flow table contains rules ordered by their priority. Each flow rule defines a matching condition, and an *action* taken if the packet header satisfies the rule criteria. During the processing of a single packet, these actions can modify the packet, forward it out of the switch, or drop it. The number of tables, the set of match fields and the set of actions that a switch can take are all fixed in the standardized data plane.

Rules are tested in decreasing order of priority until a matching rule is found. A packet arriving at a switch is first matched against the rules in the first table and assigned a set of actions to be applied at the end of the table processing pipeline. The instructions in a matching rule in any table can choose to manipulate this set and can also apply some actions to the packet before it is processed by the next table in the pipeline. Each flow rule has two parts:

- **Match:** A set of packet header field values that apply to the given rule. Some are characterized by single values (e.g. VLAN ID: 1, or TCP Destination Port: 80), others by a range (e.g. Destination IP

Addresses: 10.0.0.0/8). If a packet header field is not specified then it is considered to be a wildcard.

- **Instructions Set**: A set of actions applied by the flow rule to a matching packet. Besides specifying a single output port for the packet, the actions can specify a variety of parameters to perform fast-failover and/or allocation of bandwidth to a particular flow. These parameters include:
  - **List of Watch and Fail-over Ports**: The switch can specify a list of tuples of watch and output ports. The switch monitors the liveness of the watch ports. The packet is sent out using the first live port in the list.
  - **Queue References**: Every OpenFlow switch is capable of providing isolation to traffic from other flows by enqueuing them on separate queues on the egress port. Each queue has an associated QoS configuration that includes, most importantly, the service rate for traffic that is enqueued in it. The OpenFlow standard itself does not provide mechanisms to create new queues and assumes that they are predefined by the underlying architecture (e.g., hardware chip); however, each flow rule can refer to a specific queue number for a port, besides the OutputPort.
  - **Meters**: Beyond the isolation provided by using queues, OpenFlow switches are also capable of limiting the rate of traffic in a given network flow by using objects called meters. Meters on a switch are stored in a meter table and can be added/deleted by using messages specified in OpenFlow specification. Each meter has an associated metering rate. Each flow rule can refer to only a single meter.

Programmable Data Plane

The P4 ecosystem has been gaining traction in both academia and industry for implementing novel data-plane functions. It comprises an open-source $P4_{16}$ language [20] [45] and the accompanying Portable Switch Architecture (PSA) [46]. The ecosystem has accelerated the design and adoption of novel network functions by enabling fully programmable data-planes without compromising the line-speed performance of modern network devices.

Furthermore, the primary goal of the P4 ecosystem is to make data-planes programmable by allowing expression of per-packet computations performed on a network device. As such, it is not designed to enable network coding applications. However, it does provide several features that make it well-suited as a platform for implementing linear NC functions for failure resilience and multicast rate enhancements. Unlike an OpenFlow switch, a switch with a programmable data-plane contains separate table processing pipelines at the ingress and egress. The pipelines are identical and the illustration in the bottom of Figure 2.1 shows a simplified example of a pipeline in the PSA. Below, we briefly describe some of the relevant features:

- **Programmable Parsing**: Unlike a standardized data plane, the programmable data-plane can parse arbitrary packet headers and extend them as needed without needing to change the hardware. As illustrated in the bottom of Figure 2.1, each arriving packet is subject to a programmable parser (left-most block) and corresponding deparser (right-most block) immediately before it leaves.

- **Customizable Packet Processing Pipelines**: There are separate ingress and egress pipelines on each PSA device which can be configured from the control-plane. These pipelines are constructed using tables. Each table has a set of fields (called its `key`) that determines the packets that *match* it. Each table also has associated C-like sub-routines called actions. The actions can perform nearly arbitrary operations on the packet headers including for example addition, multiplication and XOR.

- **Packet Cloning & Recirculation**: Cloning makes copies of packets on the egress pipeline, while recirculation sends the packets from the egress pipeline to the ingress pipeline. Both of these features can be used in tandem to *generate* a new packet for coding/decoding operations. Furthermore, since P4 does not have a primitive analogous to a loop in imperative programming languages, packet cloning and recirculation can also be used to create one without any intervention from end-hosts.

- **Registers**: Registers are essentially global variables that can hold global state independent of any specific packet in any given pipeline. These registers can be used to drive state machines and implement data structures that hold packets that are required to be coded.
- **Extensibility**: P4 allows extension of the core language by using a construct called `extern`. Essentially, this construct allows another level of flexibility to implement features that do not exist in the language. Such flexibility may be crucial to any implementation of NC that goes beyond simple linear codes.

2.2.3 Southbound Interface

The interface between control and data plane is called the southbound interface. It is essentially a standardized [19] unicast TCP connection between each switch and the controller. It is a bi-directional interface: the controller sends instructions to switches and also receives current state from the switches. Such state includes interface and flow counters. The controller and switches also exchange periodic messages to establish liveness of the other node. The switches also send link-layer updates (e.g. change in a link status etc.) to the controller as needed.

The semantics of the information exchanged between the data and control planes have evolved. When the first version of OpenFlow was released, the focus was on demonstrating a proof-of-concept and hence the semantics were simple: The controller sent flow rules down into a single table inside the switch and read counters from the switches. However, with programmable data-planes, the interface now includes capabilities to re-program the behavior of the switch via the controller itself [44]. This includes changing how many flow tables are available at the switch.

2.2.4 Northbound Interface

The interface between the individual control-plane application and the consolidated control-plane function is called a northbound interface. It is exposed by the controller. There is no standardization around the semantics of the northbound interfaces due to the variations in the functionalities of the individual controllers. However, these interfaces are typically built on top of the secure HTTP protocol, and the use of specially designed application layer protocol is avoided to allow the application logic to be written in a wide variety of programming languages.

The northbound interface allows innovation in programmable network, helping application designers to try new ideas without re-inventing the entire control-plane software stack. Each design tool that we propose in this dissertation uses the northbound API to perform different functions.

CHAPTER 3

ANALYSIS: ACCESS AND CONTROL AND RESILIENCY GUARANTEES

As described in Chapter 2, the programmable control plane provides flexible switch-level mechanisms to respond to link and device failures. However, the network-wide effect of the usage of these mechanisms may lead to violation of the administrative policies governing the smart grid network. In order to guarantee that a network satisfies its requirements, its control-plane state must be validated. Such validation requires careful modeling of the control-plane state of the network and the impact of the failure events. The automatic validation mechanism needs to take into account all possible combinations of packet header bits and the actions performed on them by each switch. This can result in a very large state-space. While the prior work [47] [48] has proposed state-space efficient abstractions for the analysis, it does not provide mechanisms to simultaneously validate resiliency and security properties of a given network. To that end, we propose an automated validation mechanism that tractably validates the state of an network. Our contributions are:

- A resilient routing policy (RRP) specification to express requirements for smart grid communication networks that use fast failover packet egress action. This specification allows simultaneous expression of requirements pertaining to the resiliency and the security properties of a network. We demonstrate the use of RRP with case studies in the context of networked micro-grids and substations.
- Models that represent a snapshot of the network state and the packet flows enabled by it. These models are extensions of our previous work [49] and allow simultaneous exhaustive packet header and link failure contingency analysis.
- Mechanisms to perform validation of policies expressed as RRP. These mechanisms divide the validation in two phases of offline pre-computation

of analysis state and subsequent querying of this state with validation queries. Such division ensures that multiple queries pertaining to the same snapshot of the network state are answered without performing the exhaustive analysis from scratch, hence resulting in amortization of validation time.

The rest of this chapter is organized as follows: Section 3.3 describes the resilient routing policy specification with a concrete example; Section 3.4 describes our model; Section 3.5 defines data structures and algorithms that implement our model and the policy validation mechanisms; Section 3.6 evaluates the performance and viability of performing exhaustive traffic analysis; Section 3.7 demonstrates the use and scalability of our validation mechanism using case studies.

3.1 Related Work

In traditional distributed control-plane networks, the errant behavior of a network device is typically rooted in misconfiguration or a bug. `Fireman` [50] is a firewall-specific static analysis tool that allows checking for inconsistencies in implementation of access control. `Anteater` [51] and `Libra` [52] illustrate requirements validation by using only the data-plane state from campus and data-center networks respectively. These tools parse the configuration and the data-plane state of heterogeneous devices; thus, individual device models become bottlenecks to predicting overall network behavior. The configuration languages on these devices evolve and these devices also interact with each other using non-compatible, bug-ridden implementations of protocols.

While the SDN architecture addresses some of the problems faced by the distributed control-plane networks, the flexibility to program individual flows can lead to behaviors that do not conform to the administrative policy. Kang et al. [25] proposed that the entire network to be viewed as a big switch and specification of policy for endpoints and paths that a subset of traffic takes through the network. Kazemian et al. proposed that the network be viewed as a rules graph and specification of policy in terms of traffic generators and receivers by using a specialized language called $flowexp$ [53]. However, neither of these semantics allow specification of behavior of the network when the topology of the network changes.

For policy validation, Kazemian et al. developed an abstraction to represent a set of packets called Header Space [47]. They developed tools for verification of requirements such as reachability, access control and loop detection [53]. Similarly, Khurshid et al. developed an abstraction for representing commonalities in the traffic referred in the flow rules as trie-based data structures [48]. However, both of these approaches validate an SDN comprising of switches that are not capable of the fast failover mechanism. This mechanism was added to the switch specification in the OpenFlow version 1.3 [19] and allows choosing alternative ports for traffic on a switch when a link fails without controller intervention. Our previous work [49] proposed the port graph model that incorporates this mechanism and is agnostic to a given switch specification. We extend this model and provide mechanisms that anticipate the impact of the failure events and thus avoid any recomputation of analysis state for a specific policy.

Finally, instead of validating the control-plane state of an SDN against a stated policy, there are proposals for synthesizing the control-plane state that can provide certain resiliency guarantees in a smart grid network by using the fast-failover mechanism. Proposals by Sharma et al. [54], Aydeger, et al. [55] and Gyllstrom et al. [56] use fast failover rules to provision paths that have backups. However, these proposals do not consider provisioning of simultaneous security guarantees.

3.2 The System Architecture

In order to perform exhaustive analysis, our approach requires access to the logically centralized *state* of the network: the network topology and the forwarding rules installed on switches. In the programmable networking architecture, the controller is the purveyor of this state and makes it available via a *northbound* API to be used by the applications. As illustrated in Figure 3.1, our prototype (called the Flow Validator) uses the northbound API to access a current snapshot of the network state.

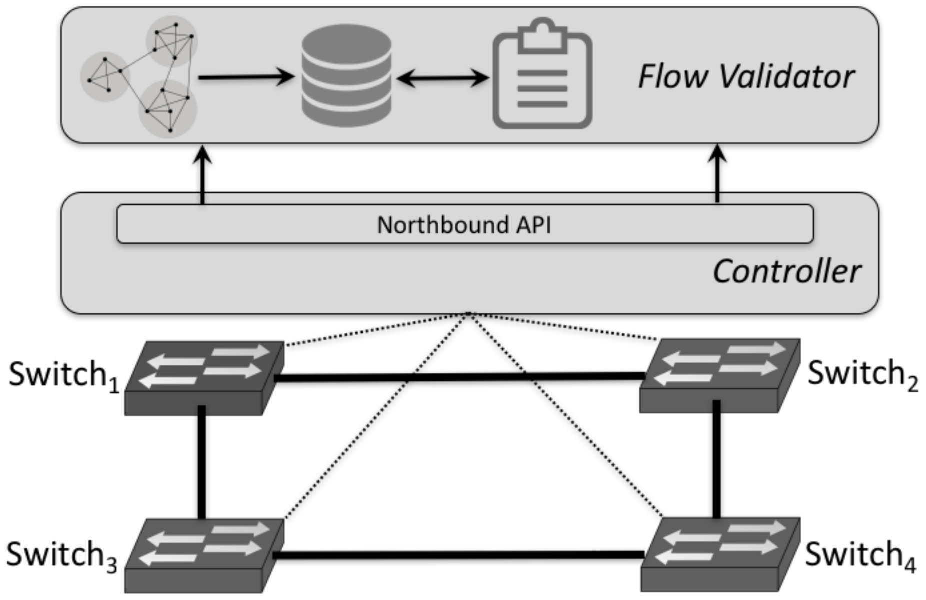


Figure 3.1: A programmable network containing four switches connected in a ring. Each switch also connects to the controller via a management port (dashed line).

3.3 Resilient Routing Policy (RRP)

The cyber network for power grids is subject to unique, simultaneous requirements of security and resiliency for access to the critical assets. Thus, in order to state such requirements for the network supporting a power grid, we propose a policy specification called *resilient routing policy* (RRP).

The RRP refers to the elements of the network topology such as ports, links and switches. Each switch s_i is assigned a number i. A physical link between the switches s_i and s_j is denoted $l_{i,j}$. Each port $p_{i,j}$ refers to the port number j on s_i. Formally, a policy P is a set of statements of expectations from the network under consideration. Each statement specifies some aspect of the intended behavior of the network. Given a set of events e, each policy statement refers to a source zone (z_i), a destination zone (z_j), a set of packets it pertains to (t) and a set of resiliency constraints (c) that are required to be validated. Thus a policy is:

$$P = \{s_1, s_2, ...s_n\}, \quad \text{where}$$
$$s_i = (z_i, z_j, t, e, c) \quad \text{for } i = 1, \ldots, n,$$

with further explanation of zones, traffic zones, events, and constraints below.

Zones (z_i, z_j)

A zone is a set of ports that a policy statement refers to. Recall the notation that $p_{i,j}$ specifies a port on switch number i with port number j.

Traffic Set (t)

The traffic set is a set of field-value pairs (i, v_i), where i is the field number in the packet header, and each v_i is a set of integer intervals, where each interval is a pair of low and high values given by $[l_j, h_j]$. Thus, in a packet header containing d fields, a traffic set is specified by:

$$\begin{aligned} v_i &= \{[l_0, h_0], [l_1, h_1], ...[l_j, h_j]\} \\ t &= \{(i_1, v_1), (i_2, v_2), ...(i_d, v_d)\} \end{aligned}$$

Events Set (e)

The policy statement contains a set e of permutations of links. The policy applies to the failure of each of these permutations of links, one at a time. For validating against failure of a switch, the permutation comprises all the adjacent links to the given switch.

Constraints Set (c)

As each failure events in the network occur, the flow rules using fast-failover will choose alternative paths for the traffic. The Flow Validator exhaustively analyzes resulting traffic flows after each event. These traffic flows are compared with the expectation of a policy statement. Each policy statement defines a set of constraints, c, where each member represents one of the following orthogonal requirements for the packets in traffic set t from a source zone to the destination zone. Following is a list of constraints that can be specified about the given network configuration:

- **Connectivity** C**:** Asserts that the configuration enables packets to flow between the source and destination zones.

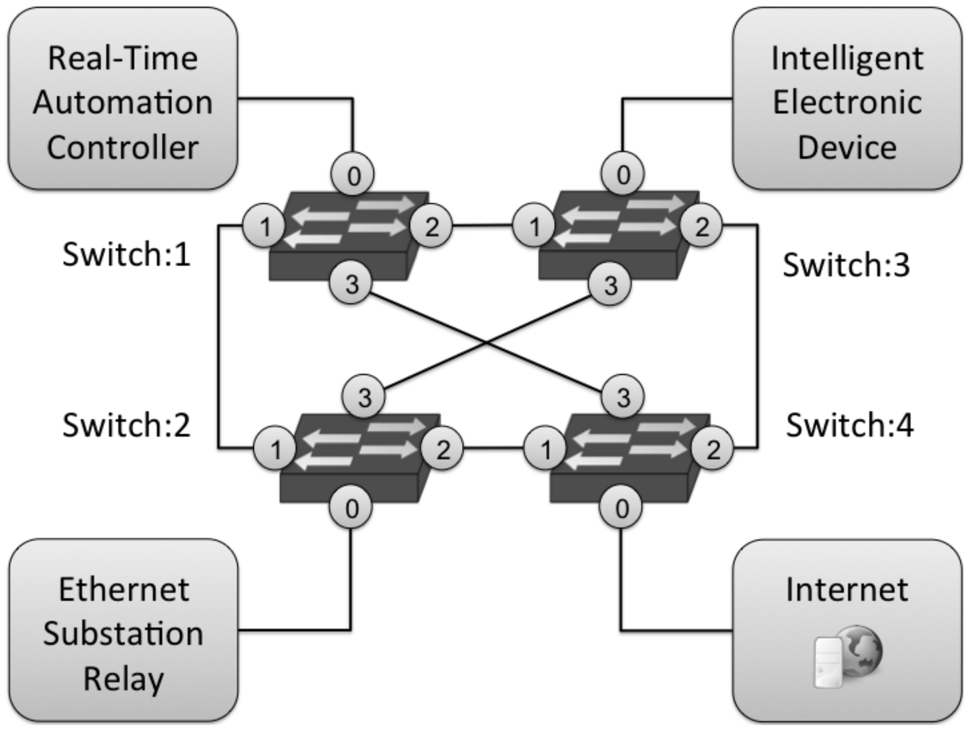

Real-Time Automation Controller
Intelligent Electronic Device
Switch:1
Switch:3
Switch:2
Switch:4
Ethernet Substation Relay
Internet
0
1
2
3

Figure 3.2: A substation network topology containing four switches. The clique provides alternative paths that may be used by the fast-failover mechanism. The users can log in via the Internet and connect to RTAC to perform configuration and maintenance tasks.

- **Isolation** I**:** Asserts that the configuration blocks the packet flow between source and destination zones.
- **Path Length** L_N**:** Asserts that the paths taken by the packets from the source zone to the destination zone have a specified maximum path length (N). The length is measured in the number of switches in the path.
- **Link Avoidance** A_L**:** Asserts that the paths taken by the packets from the source zone to the destination zone do not traverse a specified physical link (L) in the network topology.

3.3.1 Example Policy

In order illustrate various elements of the RRP, we present an example. The example policy below concerns the CI network inside a substation as shown in Figure 3.2.

$$z_1 = \{p_{1,0}\}$$

$$z_2 = \{p_{2,0}, p_{3,0}\} \qquad z_3 = \{p_{4,0}\}$$

$$v_1 = \{[-\infty, +\infty]\} \qquad v_2 = \{[443, 443]\}$$

$$t_1 = \{(i_1, v_1) \ldots (i_d, v_1)\} \qquad t_2 = \{(i_1, v_1), (i_2, v_2), \ldots (i_d, v_1)\}$$

$$c_1 = \{C, L_3\} \qquad c_2 = \{C, A_{l_{3,4}}\}$$

$$e_1 = \{(l_{1,3}), (l_{3,4}), (l_{4,2}), (l_{2,1}), (l_{1,4}), (l_{2,3})\}$$

$$s_1 = (z_2, z_1, t_1, c_1, e_1) \qquad s_2 = (z_3, z_1, t_2, c_2, \emptyset)$$

$$P = \{s_1, s_2\}$$

Tbe policy defines three zones: z_1, z_2 and z_3, each containing the switch ports attached to the Real-Time Automation Controller (RTAC), the Ethernet Substation Relay (ESR) and the generator Intelligent Electronic Device (IED), and the Internet gateway port respectively.

By using the zone definitions above, the policy statement s_1 specifies what occurs to the packets from z_2 to z_1 when any single link in the topology fails. In order to concern every packet, it uses a traffic set t_1 which covers the entire range of the interval for each field in the packet header. To specify the failure events it concerns, it uses an events set e_1 which contains every permutation of a single link failure in the topology. Finally by using the constraint set c_1, it asserts that even after any event in e_1 occurs, the network continues to provide connectivity for this traffic such that the maximum path length is three switches (i.e. $N = 3$).

Similarly, the policy statement s_2 specifies access control for traffic from z_3 to z_1. It only allows the SSH traffic by using a traffic set t_2 to represents the packets with the destination TCP port 443 in the header. It does not concern any specific failure events but asserts that the network provides connectivity for this traffic such that its path does not traverse the link between switches s_3 and s_4 (i.e. $L = l_{3,4}$).

3.4 Model

In order to validate policies expressed using RRP, we propose a model to represent the control-plane state of programmable network using three separate but inter-linked abstractions. The first abstraction models the packet forwarding and modification decisions of the control-plane state. The second abstraction models the sets of packet headers that are delivered from any one port to another in the network as a result of the control-plane state. Finally, the third abstraction models the exact sequence of intermediate switch ports that the delivered packets traverse. Analysis of these models reveals precisely what traffic is permitted, descriptions against which we determine whether the permitted traffic always complies with policy. Here, we describe each one of these abstraction in detail.

3.4.1 Port Graph

The port graph model represents the control-plane state and the physical topology of a given network with a directed graph (N, E). The set of nodes (N) in a port graph models the points of interest in a network through which traffic can flow. These points are either the physical ports on the switches, or flow tables where computation (e.g. packet modification, forwarding decisions) occurs. The set of edges (E) models the transfer of traffic from their predecessors to successors.

The model uses port graphs on two levels. On the lower level, a port graph models switch transfer function originating from the switch's configuration as described in Chapter 2. On the higher level, a port graph models the entire network and is constructed using the transfer functions of the individual switches and the network topology. Such distinction between two levels of port graph offers two advantages: First, it makes the model independent of the switch specification. So, if a network is composed of heterogeneous switches running different OpenFlow versions or standards, it can still be modeled using their individual port graph. Second, it provides algorithmic

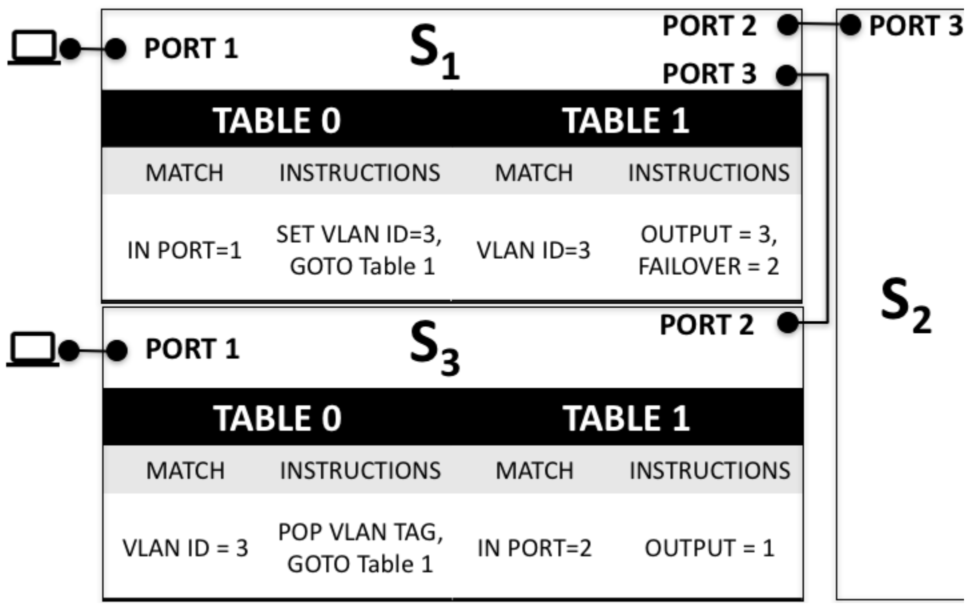


Figure 3.3: A network comprising of three switches. The switches (s_1 and s_3) have two flow tables and one host each.

efficiencies by reducing the number of nodes and edges that are present in the higher-level network port graph.

Figures 3.4 and 3.5 illustrate the port graphs on both levels for a simple three-switch topology shown in Figure 3.3. Below we describe what constitutes the port graph's nodes and edges on each level:

Nodes

Nodes model the points of decision. Every switch in the SDN has multiple nodes in the port graphs on both levels. Every switch port is represented with two nodes, one for each direction of traffic: ingress/egress. In addition to these nodes, every flow table in the switch has a node in the port graph as well. Hence the total number of nodes per switch in the port graph is $2 * p + t$, where p is the number of switch ports and t is the number of tables in the switch.

Edges

Edges model the transfer of traffic from predecessors to successors. Each edge has two attributes associated with it: An edge filter $EF_{(p,s)}$ that intersects

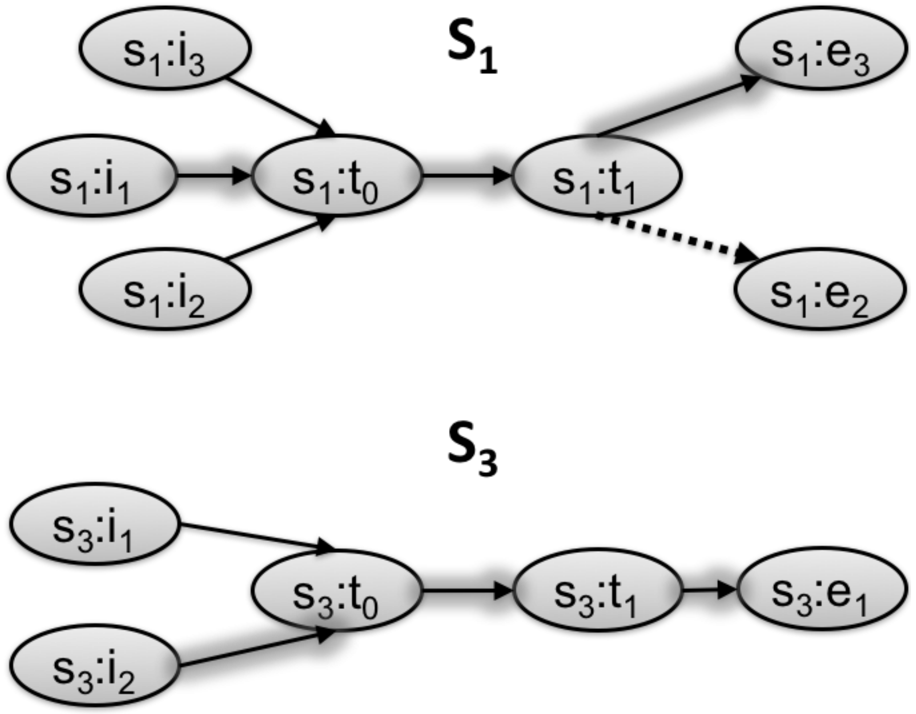


Figure 3.4: The switch port graphs representing the transfer function of s_1 and s_3. The dashed edge represents an inactive edge.

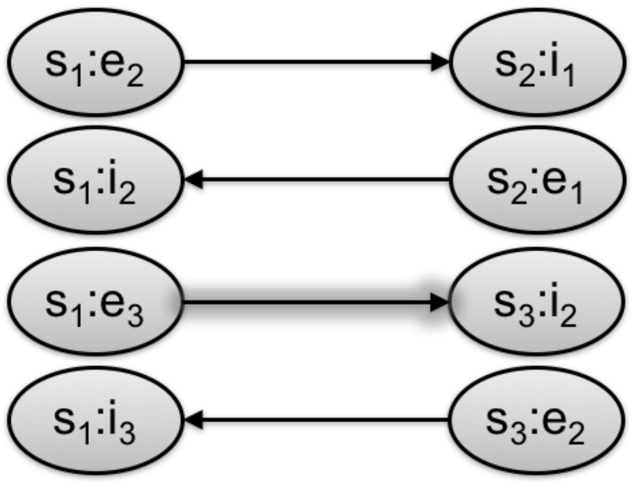


Figure 3.5: The network port graph.

as a *match* with the traffic transferring from predecessor node p to successor node s and a list of fields that are modified in this transfer. There are three types of edges in the switch port graph:

- Packet Arrival Edges: These edges model the start of packet processing for the switch. Packets are first examined by flow rules in the first flow table of the switch, thus resulting in edges from the ingress nodes representing each port to the node representing the first flow table.
- Packet Departure Edges: These edges model departure of packets due to an action applied by a matching flow rule in a flow table. These are edges from table node representing the flow table containing the rule to the node representing the egress node for the port. Since rules in a flow table are matched in the order of priority, there is an intra-table dependency between traffic flows that matches a flow rule. This is illustrated in Table 3.1. Consider traffic T arriving at a flow table containing flow rule F_1. Packets that match this rule are not matched with the next rule F_2 and so on.

Table 3.1: Flow table model with n rules and T as input traffic set

Input Traffic	**Flow Rule Match**	**Flow Edge Filter**
T	F_1	$T \cap F_1$
$T \cap {F_1}^c$	F_2	$T \cap {F_1}^c \cap F_2$
$T \cap {F_1}^c \cap {F_2}^c$	F_3	$T \cap {F_1}^c \cap {F_2}^c \cap F_3$
...	...	...
$T \cap {F_1}^c \cap {F_2}^c \cap ...{F_{n-1}}^c$	F_n	$T \cap {F_1}^c \cap {F_2}^c \cap ...{F_{n-1}}^c \cap F_n$

- Inter-table Edges: These edges model the transfer of packets from one table to another in the pipeline. Each *goto-table* instruction found in a flow rule results in an edge from the node representing its flow table to the node representing the table that the instruction refers to.

Furthermore, we distinguish between modifications that are applied immediately before a packet arrives at a flow table in the processing pipeline, and those applied at the end of pipeline. We also attribute a flag to each edge in the switch port graphs resulting from fast failover actions in a flow rule. The flag indicates whether a given port graph edge is active. Finally, while not shown in the figures, we also capture any packet modifications by tracking the `set-vlan` actions in the configuration.

There are two types of edges in the network port graph:

- Physical Link Edges: These edges represent the bi-directional flow of unmodified packets across each physical link in the topology. Thus, the edge filter is a wildcard with no field modifications.

- Switch Transfer Function Edges: These edges are a direct consequence of the output of the switch port graph of the a switch. For a given ingress and egress port node pair (i, j), if there is admitted traffic in the switch port graph, then an edge is added in the network port graph between i and j. The filter for this edge is $ATS_{i,j}$ in the switch port graph. The modifications for this type of edge are an accumulation of all the modifications that the switch port graph applies to that traffic as it goes from ingress to egress port node.

3.4.2 Admitted Traffic Set (ATS)

An *admitted traffic set* ($ATS_{p,d}$) in the network port graph is defined as representing the traffic carried from a source node p to the destination node d, where $p, d \in N$. This definition makes it so that the nodes that represent the ports connecting the physical links become the intermediate nodes and are not explicitly represented by the $ATS_{p,d}$.

For example, with reference to the Figure 3.3, assume that one is interested in the $ATS_{p,d}$ from the port 1 on the Switch s_1 to the port 1 on the Switch s_3. Here, port 3 on the Switch s_1 and port 2 on the Switch s_3 are the intermediate ports represented with their ingress and egress nodes in the port graphs in the Figures 3.4 and 3.5. However, the $ATS_{p,d}$ shall not concern these intermediate ports.

The reason the intermediate nodes are not explicitly represented is that, due to the use of fast-failover actions, the intermediate nodes may change based on the status of the links. However, our model separates the exact *path* from the node p to d from the definition of the traffic itself.

3.4.3 Active Traffic Path

A traffic path is a sequence of ports that the traffic in the set $ATS_{p,d}$ traverses. However, as noted above, this path may change based on the status of the links in the topology. We assume that at any given time, there is at most a single path from a source p and a destination d. Given a set active links, the resulting traffic path is defined as the *active* traffic path. The existence of an active path between a node-pair is a direct consequence of the configuration (i.e. flow rules) and the topology of the network.

The active traffic path is represented as a sequence of edges from the source node p to the destination node d. For an edge to be part of a traffic path, it is necessary that it *enables* the traffic at its predecessor node to be carried to its successor nodes after filtering or modification at its predecessor node. For example, the highlighted edges in Figure 3.4 represent the path from the host at the Switch s_1 to the host at Switch s_3. Furthermore, since only one of the edges is active per matched flow rule, there is only one path composed entirely of active edges.

3.5 Design

Our design makes the choice of dividing the total computation required for policy validation in two phases: We first precompute traffic flows through the network assuming that a traffic wildcard were to appear at switch ports. We then use this precomputed state to validate specific instances of policies expressed using RRP. This choice allows amortization of the cost of exhaustive analysis over many policy validation queries.

First, the design requires the construction of the switch and network port graphs as described in Section 3.4.1. In this section, we describe the data structures and algorithms that we use to precompute and store the state to enable the policy validation. Finally, we describe the algorithms to perform policy validation itself.

3.5.1 Precomputation of Admitted Traffic Sets

Before the computation of $ATS_{p,d}$, the switch transfer functions are initialized. This initialization is done by using a breadth-first-search that carries and initializes the $ATS_{p,d}$ for each node of the switch port graphs. The search starts at every possible destination node (i.e. port egress nodes) and starts with carrying a wildcard traffic set. At each node, it explores all the predecessors of the node and modifies the traffic set when each edge in the port graph is traversed. The modification takes into account the edge filter and the modifications that are applied at the predecessor of the edge. The search terminates at the port ingress nodes because they do not have any predecessors.

Once the traffic is propagated on the switch port graphs, $ATS_{p,d}$ is computed. If the source and destination nodes are on the same switch, the switch's transfer function is directly referred. Otherwise, the computation is handled in three phases. First, we identify the pairs of intermediate nodes, with the egress node at the source switch and ingress node at the destination switch, that carry a subset of the $ATS_{p,d}$. Next, we intersect the traffic from source node to intermediate egress node with this subset. Lastly, the result of this intersection is intersected with traffic from intermediate ingress node to the destination node to obtain the subset of $ATS_{p,d}$.

3.5.2 Computation of Active Paths

Algorithm 1 returns one or more edges as the path AP by using a depth-first search. The search starts at a source node and ends at the destination node if no loops are detected. It proceeds recursively, following the admitted traffic sets along the port graph edges. This algorithm relies upon other auxiliary methods; for example, in line 3, it uses a method called CanTraverseEdge to check if a given port graph edge is active and if its edge filter allows a subset of $ATS_{p,d}$. This requires an intersection operation and checking a flag. Similarly, in line 8, it uses a method called PathHasNoLoops, which uses cycle-detection to check if the path accumulated thus far has cycles in it. If there are no such cycles, then the edge is appended to the path and a recursive call is made in lines 9-10. Line 4 stops the said recursion at the destination node.

Algorithm 1 Computation of the paths from a node p to destination d that carry traffic $ATS_{p,d}$.

Input: The source and destination nodes p and d and the admitted traffic set $ATS_{p,d}$ from p to d, the set of successors for each node in the graph $\sigma_{p,d}$ and an empty list AP.

Output: The active path AP is populated with the edges from p to d.

```
1:  function GetActivePath($p$, $d$, $\sigma_{p,d}$, $ATS_{p,d}$, $AP$)
2:      for all $s \in \sigma_{p,d}$ do
3:          if CanTraverseEdge($p$, $s$, $d$, $ATS_{p,d}$) then
4:              if $s == d$ then
5:                  $AP$ =Append($AP$, $E_{p,s}$)
6:                  return
7:              else
8:                  if PathHasNoLoops($AP$, $s$) then
9:                      $P$ =Append($AP$, $E_{p,s}$)
10:                     GetActivePath($s$, $d$, $\sigma_{p,d}$, $ATS_{s,d}$, $AP$)
11:                 end if
12:             end if
13:         end if
14:     end for
15: end function
```

While the run-time of this algorithm is seemingly bounded by the run-time of the depth-first search, there are several ways in which exact composition of the flow rules in the underlying network configuration affects the run-time. For example, if the match of a flow rule is very specific, then a relatively simple traffic set is carried to the next edge filter. Furthermore, if a flow rule uses fast-failover mechanism, then it results in multiple edges, thus causing the search in Algorithm 1 to take longer. Finally, the order of matching flow rules in the flow tables matters because the flow rules at the bottom of the table have a more complex edge filter than those at the top.

3.5.3 Computation of Failover Paths

Algorithm 2 Computation of the failover path that carries traffic $ATS_{p,d}$ when the links in λ have failed.

Input: The current active path AP and a sequence of links λ.

Output: The failover path FP, if one exists after the links in λ are presumed to have failed or an empty list otherwise.

```
1:  function GetFailoverPath($AP$, $\lambda$)
2:      $FP = AP$
3:      for all $l \in \lambda$ do
4:          SetLinkPortsDown($l$)
5:          $E_{p,s}$ = GetAffectedEdge($FP$, $l$)
6:          if $E_{p,s}$ then
7:              $E^{'}_{p,s}$ = GetAlternativeEdge($E_{p,s}$)
8:              $PP$ = GetAffectedPrefixPath($FP$, $l$)
9:              $SP = []$
10:             GetActivePath($s$, $d$, $E^{'}_{p,s}.EF$, $SP$)
11:             if $E^{'}_{p,s}$ & $SP$ then
12:                 $FP = PP + [E^{'}_{p,s}] + SP$
13:             else
14:                 $FP = []$
15:                 break
16:             end if
17:         end if
18:     end for
19:     for all $l \in \lambda$ do
20:         SetLinkPortsUp($l$)
21:     end for
22:     return $FP$
23: end function
```

Algorithm 2 returns a failover path if the given active path survives the failure of a sequence of links (λ). To that end, it simulates link failures and finds alternative active paths due to the use of fast-failover mechanism by the flow rules. Specifically, it simulates link failure and restoration events in lines 4 and 18 respectively. This is done so that the method CanTraverseEdge in Algorithm 1 described previously can correctly ascertain the path of traffic across the affected port graph edges. Once the failures have been simulated, the restorations are also simulated before exiting the algorithm, so that the next call results in correct state of the link.

The algorithm proceeds with simulating the failure of one link at a time by iterating over λ. The loop from lines 3-17 performs the key operation of finding an alternative path in case of each link's failure, if one exists. First, in line 5, the algorithm checks if the failed link affects the primary path at all by using the auxiliary method GetAffectedEdge. If the failure does affect the path, then in line 7, the algorithm uses the method GetAlternativeEdge to find an alternative port graph edge to the affected port graph edge. If an alternative edge exists, then in line 10, the algorithm finds the remaining active path from the successor of the alternative edge to the destination. If such a path from the successor exists, it means the path survives the link failure of the particular link in the sequence. The exact path is reconstructed in line 12 by concatenating the path until the affected edge (obtained using another auxiliary method GetAffectedPrefixPath), the alternative edge and the remaining active path.

However, if the presumed failure of any of the links in λ results in no remaining active path SP or if there is no alternative edge to the affected edge, then it means that there is no failover path for the given λ and consequently the algorithm returns an empty list in line 14.

3.5.4 Policy Validation

After the precomputation is done, the user prepares the RRP policy they want to validate. It is posed to the Flow Validator as a policy validation query. Each query contains a set of policy statements in the form described in Section 3.3. Each query results in the analysis of the effect of failure events on the $ATS_{p,d}$ and active traffic paths.

Before the validation begins, the policy query is first transformed into an equivalent and representative data structure called $PMap$. The goal of this transformation is to avoid repetition of validation even when the policy query may contain multiple references to the same combination of sequence of link failures (λ) and a source and destination node pair (s_p, d_p). Therefore, the

Algorithm 3 Policy Validation algorithm.

Input: A nested policy hashmap $PMap$ representing the contents of the policy query.

Output: The set v containing all the policy violations found.

```
1:  function ValidatePolicy($PMap$)
2:      $v = \emptyset$
3:      for all $\lambda \in PMap$ do
4:          for all $(s_p, d_p) \in PMap[\lambda]$ do
5:              for all $s \in PMap[\lambda][(s_p, d_p)]$ do
6:                  $AP =$GetActivePath($s.t$, $(s_p, d_p)$)
7:                  $FP =$GetFailoverPath($AP$, $\lambda$)
8:                  for all $c \in s.Constraints$ do
9:                      if $c == Connectivity$ & $\neg FP$ then
10:                         Append($v$, $(\lambda, s_p, d_p, c)$)
11:                     end if
12:                     if $c == Isolation$ & $FP$ then
13:                         Append($v$, $(\lambda, s_p, d_p, c)$)
14:                     end if
15:                     if $c == Path\ Length$ & $FP$ then
16:                         if $FP.NumSwitches > c.N$ then
17:                             Append($v$, $(\lambda, s_p, d_p, c)$)
18:                         end if
19:                     end if
20:                     if $c == Link\ Avoidance$ & $FP$ then
21:                         if $FP.CrossesLink(c.L)$ then
22:                             Append($v$, $(\lambda, s_p, d_p, c)$)
23:                         end if
24:                     end if
25:                 end for
26:             end for
27:         end for
28:     end for
29:     return $v$
30: end function
```

$PMap$ is a two-dimensional hashmap. The first and second dimensions of the key of $PMap$ are λ and (s_p, d_p). The corresponding values to each key in the $PMap$ are a list of statements referring to the λ and (s_p, d_p) in the key.

We use Algorithm 3 to find all the policy violations in a given $PMap$. Line 2 initializes the set v to an empty set. Lines 3-5 start three loops that iterate over the contents of the $PMap$. This results in exploring every combination of the sequence of link failures (i.e. λ), the node pair (i.e. (s_p, d_p)), and the relevant policy statement s. The core policy validation for each such combination is performed in lines 6-25. First, we obtain the primary and failover paths in lines 6 and 7 respectively. We then iterate over each constraint in the policy statement s and record any violation by appending to the set v in lines 8-25. Each violation is a tuple specifying the source, destination node pair (s_p, d_p), sequence of link failures (λ) and the specific constraint (c) which has been violated.

The policy validation for each constraint, assuming the links in λ have failed, proceeds as per the specification described in Section 3.3. Line 9 checks whether the failover path does not exist and if so, line 10 records a *Connectivity* constraint violation. Line 12 checks whether a failover path does exist and if it does not, line 13 records an *Isolation* constraint violation. Line 16 checks if the length of the failover path exceeds the length specified in the constraint (i.e. N) and if so, line 17 records a *Path Length* constraint violation. Line 21 checks if the failover path crosses a link that is to be avoided per the constraint (i.e. L) and if so, line 22 records a *Link Avoidance* constraint violation.

3.6 Evaluation

In this section, we evaluate the performance of the algorithms presented in Section 3.5. In particular, we seek to characterize and compare the time it takes to perform precomputation of admitted traffic and then use the results of the precomputation to answer specific policy validation queries. We first describe our experimental setup for the empirical evaluation. Specific details regarding each aspect of our design are presented subsequently.

3.6.1 Experimental Setup

We implemented a single-threaded prototype in `cPython` called Flow Validator. We performed all the experiments on a virtual machine with 16 GB of RAM and 2 processor cores clocked at 3.3 GHz. We used `mininet` [57] to emulate the network topologies containing `Open vSwitch` [58] as the switch implementation. We used an open source controller called `Ryu` [59] and used its the northbound API to synthesize flow rules.

We performed the evaluation using four different network topologies with varying number of switches and hosts at each switch as shown in Figure 3.6. Three of these topologies (i.e. full-clique, ring and Clos [60]) are bi-connected, i.e. if any single link fails, there is still a topological path between any pair of switches. For these topologies, we used the approach described by Elhourani et al. [61] for synthesizing flow rules. These flow rules enable host-pair flow paths carrying all IP traffic even when any single link in the topology fails. The fourth topology we used is motivated by the need to provide VLAN based isolation in a network of microgrids. The details regarding the structure of this topology and how the flow rules are synthesized are described in more detail in Section 3.7. The number of flow rules required to provide connectivity for each host pair increases quadratically with the number of hosts in a given topology and the flow rule synthesis technique.

3.6.2 Initial Precomputation

We measured the time taken to perform the exhaustive analysis as the initial precomputation. This includes the time for computing transfer functions for each switch in the topology and the time of propagating traffic from each port associated with a host. The results are shown in Figure 3.7. Both axes of the plot are logarithmic. The x-axis shows the number of individual host-pairs in the topology and the y-axis shows the time taken to perform the initial precomputation.

Evidently, the time taken to perform initial precomputation increases with the number of host pairs in every topology. However, the absolute value and the trend of the relationship between the two quantities vary among different topologies. Generally, with one exception described later, the absolute value of precomputation time increases with the total number of switches

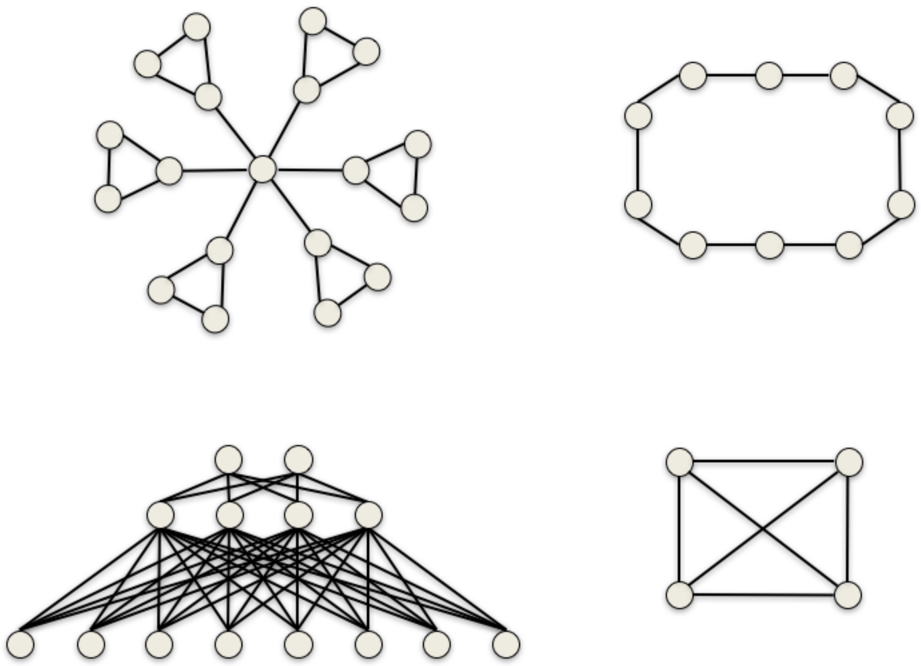


Figure 3.6: Network topologies used for evaluation, clockwise from bottom-left: Clos (14 switches), Microgrid (19 switches), Ring (10 switches) and Clique (4 switches).

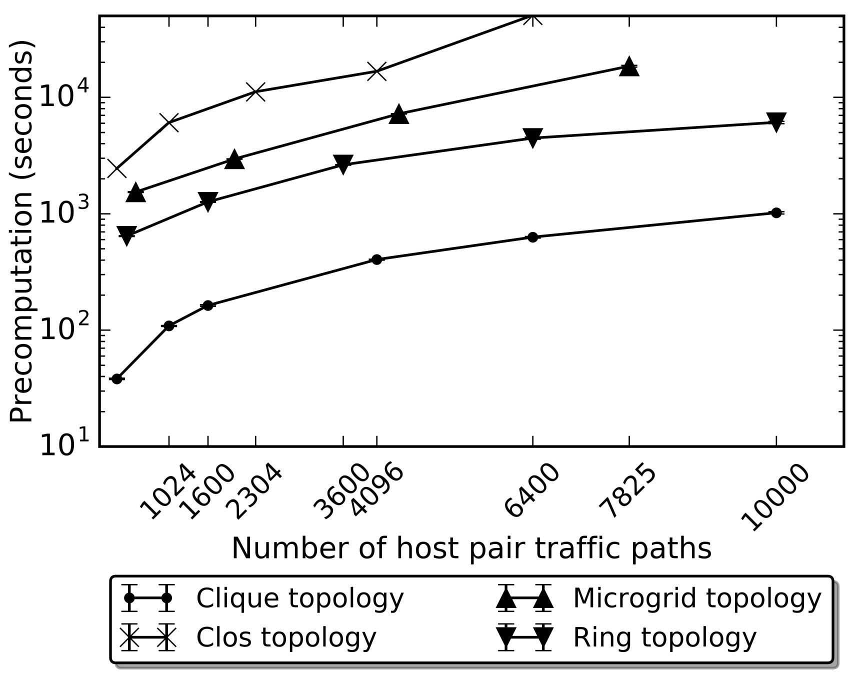


Figure 3.7: Initial precomputation for the evaluated topologies.

in the topology. This is a direct consequence of propagation in Algorithm 1. However, even though the Microgrid topology has more switches than the Clos topology, the nature of flow rules in the Clos topology is such that it results in addition of more edges in the port graph. In particular, the Microgrid topology does not have any fast-failover rules whereas the Clos topology does. Each fast-failover rule causes addition of multiple edges to model different possibilities

3.6.3 Active Path Computation

We measured the time it takes to compute a single active path using Algorithm 1. In order to measure this time, we performed validation of a policy containing a single statement. The statement specified that the length of active traffic path for each host pair in the four topologies described earlier must be less than the twice the diameter of the given topology. Here, the diameter is the longest shortest path between any two switches in the topology. The validation of this policy results in computation of the active traffic

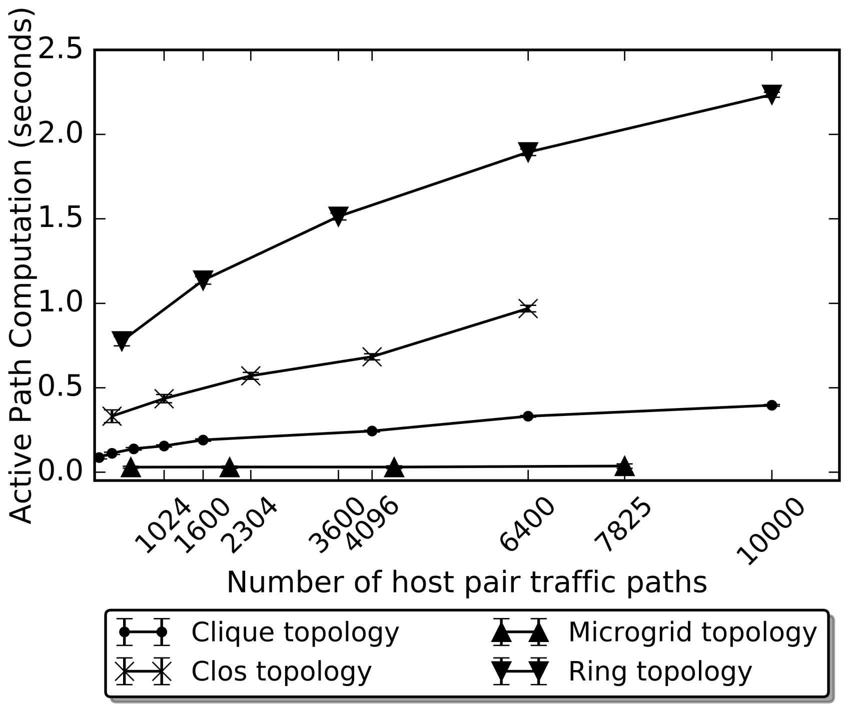


Figure 3.8: Active path computation time for the evaluated topologies.

path between each host pair in the topology, whose length is then compared to the diameter as specified in Algorithm 3. We computed the average and standard error of the active path computation time over all host pairs, and they are plotted in Figure 3.8 for the four topologies.

The average active path computation time is affected by the nature of the synthesized flow rules and the length of the paths that the rules result in. For example, the flow rules used in the Microgrid topology are such that most of them are local to each microgrid and thus are smaller in length and are computed fastest. Furthermore, the flow rules used in this topology are also very specific to an individual host, so the intersections performed by the path computation algorithms are fast. In the other three topologies, the flow rules are synthesized such that the traffic belonging to an entire switch (i.e. all the hosts at that switch) is referred with the same match at any other switch. The ring topology presents the worst case because it results in addition of many edges in the network port graph due to only two underlying physical paths.

3.7 Case Studies

In this section, we demonstrate the expressiveness of RRP and scalability of our validation mechanisms by using empirical experiments. We used `mininet` [57] to emulate two different types of cyber networks to contextualize this demonstration. The network in the first case connects hosts within a substation to each other and to the Internet. The network in the second case study connects hosts spanning multiple microgrids and the main grid. The policy statements used for each network reflect the specific set of guarantees that it needs to provide. In each case, we measure the time it takes to perform exhaustive validation of policies expressed as RRP by using the Flow Validator.

3.7.1 Resilience in a Substation Network

The cyber network inside a substation may be comprised of a set of networked control devices such as intelligent electronic devices (IED), real-time automation controllers (RTAC) and ethernet relays (ER). These host devices are considered critical assets and provisioning of the substation networks presents an adoption opportunity for the programmable networking architecture in smart grid networks. These host devices require resilient and secure connectivity for safe operation of the substation.

Consider the substation network topology depicted in Figure 3.2. The network transports IEEE C37.118 [62] packets carrying synchrophasor measurements from an IED to the RTAC. The RTAC may send control decision encapsulated in DNP3 [33] packets to the ESR. If a set of network links fail, the policy requires that all host devices continue to communicate with each other, while simultaneously maintaining compliance to the security policy. We increased the number of host devices (e.g. ER, IED or RTAC) connected to each switch to evaluate the scalability. We synthesized the flow rules for all host pairs by using the approach described by Elhourani et al. [61] to ensure that each host pair can communicate even if any single link in its primary path fails.

The number of active path computations to exhaustively validate the policy depends on both the specified number of link failures (k) and number of links in the topology ($|L|$) in the RRP instance and the topology respectively.

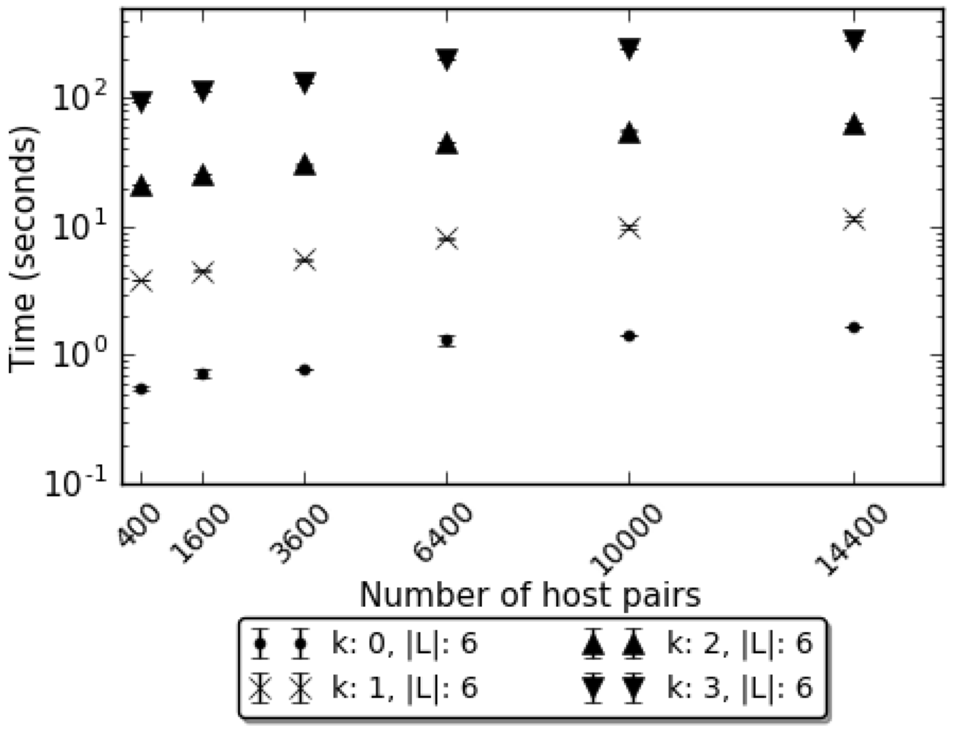


Figure 3.9: Exhaustive policy validation times as a function of k_{max} and $|L|$ and the number of host pairs in a four-switch topology in substation.

Since the topology in Figure 3.2 is a full-clique containing four switches, it follows that $|L| = 6$. However, in order to vary k, we used the same policy specification containing the two statements as illustrated by the example policy in Section 3.3, but changed the value of k for both statements to validate resiliency when up to any k links in the topology (i.e. $k_1 \in \{0, 1, 2, 3\}$) fail.

We measured the policy validation time for each value of k. We repeated each experiment for five iterations and plot the results in Figure 3.9. Note that the graph has logarithmic axes. Hence, our results indicate that the validation time increases with the number of host-pairs in the topology. This is a direct consequence of the number of host pairs that are part of the $PMap$ in the Algorithm 3.

Furthermore, the validation time is directly proportional to k, because k is proportional to the number of link permutations whose failure is to be considered for the validation. Each such permutation results in an entry in the $PMap$ and thus adds to the time taken by Algorithm 3.

3.7.2 Security for Interconnected Microgrids

A microgrid is composed of multiple energy sources and loads in a relatively small geographical area compared to the main smart grid. Multiple microgrids operate in tandem with the main grid to deliver essential services. Intelligent controllers embedded in a microgrid enable it to autonomously optimize energy generation, share energy with the main grid and isolate itself from the main grid during emergencies. Thus, controllers in a microgrid need connectivity with controllers in the main grid for coordination and stability. Regulatory regimes such as NERC CIP require tight network access control between hosts in microgrids and the main grid. Hence, the operations in the microgrid architecture require tight control over and visibility of connectivity between the hosts in any two grids.

To implement this network, we used a network architecture which overlays logical segments called *enclaves* and *functional domains* on top of the physical topology [63]. Hosts that need to communicate with other hosts within a microgrid are grouped in an enclave based on their functionality and/or physical proximity. For example, cyber nodes (IED, ER etc.) associated with a photovoltaic generator could constitute an enclave. However, inter-enclave communication among hosts grouped in a *functional domain* is allowed to enable smart grid functions that require coordination between a microgrid and the main grid. For example, an energy management system (EMS) may control the islanding operation of a microgrid by controlling a group of interconnected IEDs performing separate functions. Since each enclave and functional domain is a broadcast domain, it is implemented as a virtual local area network (VLAN). Each host inside an enclave or a functional domain is assigned to one or more VLANs reflecting its membership in enclaves and functional domains.

In our experiments, we emulated networks where one or more microgrids connect to the main grid. We used a single switch to represent the main grid control center, while each microgrid is represented by three switches connected in a ring. One of the three switches in the microgrid has a physical link to the switch representing the main grid control center. Each switch inside the microgrid ring connects to the same number of hosts given by n_h. Each switch connects to hosts from a specific enclave; thus, the number of enclaves given by n_e is same as the total number of switches in the topology.

However, we define a single functional domain containing only one host from each switch in the network. We emulated different networks by choosing different values of n_h and n_e. We synthesized flow rules for each network by using a northbound application called `ONOS-VPLS` [64].

We used Flow Validator to validate that the hosts can communicate with other hosts within an enclave or a functional domain and not otherwise. Each enclave is represented by a zone. Furthermore, we use two zones to represent hosts that are in or out of the functional domain. Each policy statement expresses the requirements for a single direction of traffic from the source zone to the destination zone. Hence, in order to validate the requirements for each enclave, we need n_e^2 statements. Similarly we need three additional statements to express requirements for traffic flowing into, out of and within the functional domain. Hence, the total number of policy statements used for each network is $n_e^2 + 3$. Finally, we used $k = 0$ for all of these policy statements, thus anticipating no resilience in the flow rules synthesized by `ONOS-VPLS`.

We measured the policy validation time for each network. We repeated each experiment for ten iterations and plot the average policy validation time in Figure 3.10 with error bars indicating the standard error (SE). Our results indicate that for a given number of policy statements, the validation time increases quadratically. This is due to the corresponding quadratic increase in the number of host pairs that are required to be validated. Furthermore, for a specific value of n_h, we observed a quadratic increase in policy validation time. This is due to the quadratic increase in the number of policy statements due to increasing n_e.

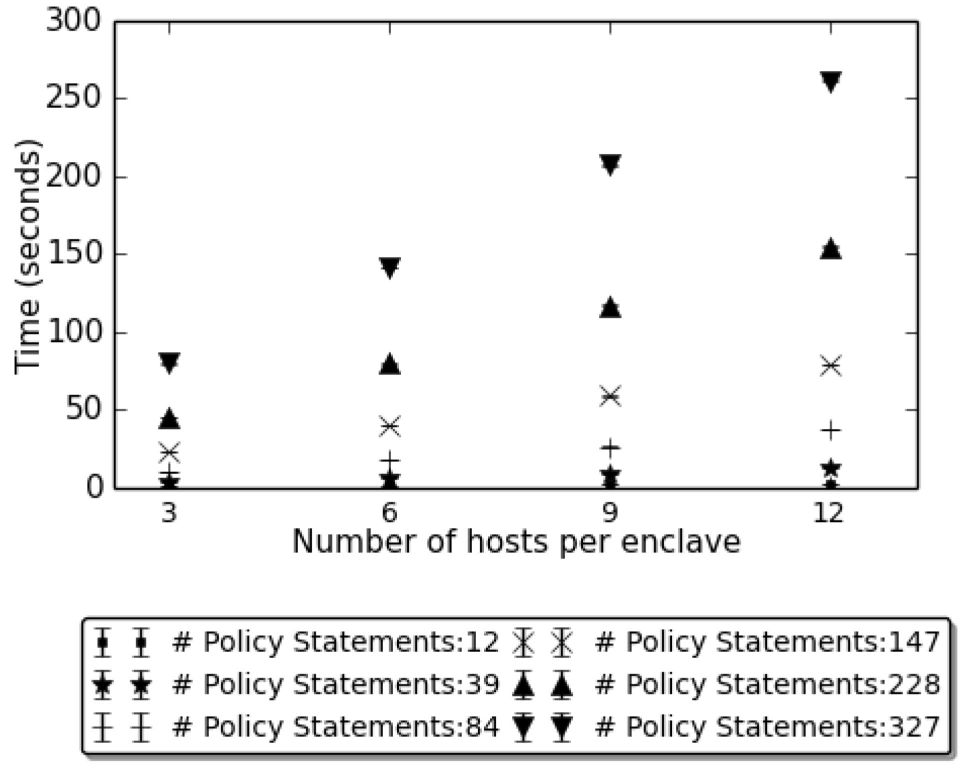


Figure 3.10: Security policy validation times for different values of n_h and number of policy statements corresponding to n_e.

CHAPTER 4

ANALYSIS: RESILIENCY METRICS

Due to the criticality of the CI systems and their absolute reliance on their cyber networks to meet their goals, the continuous operation of the CI network is paramount. However, such continuous operation requires resilience to link or network device failures. Such resilience, or lack thereof, is a consequence of the configuration of a CI network. Hence, the configuration of such networks must be analyzed to study the impact of failures on the network in a wide variety of scenarios. Such wide-ranging analysis can be effectively framed with the use of resiliency metrics.

As described in Chapter 2, fast-failover mechanism is a form of proactive configuration to provide dynamic resilience to failures. With its use, the network can be prepared to withstand a given set of link or device failures ahead of time. Its use avoids controller intervention upon link or device failures. Consequently, it results in a routing convergence time bounded by the time it takes for the switch to start using the new path. However, as the number of critical flows in a network increases, the associated fast-failover based configuration for enabling resilience for those flows can become very complex. Hence, it is absolutely critical that appropriate resiliency metrics be used to understand the consequences of using this mechanism.

There is prior work on computing resiliency metrics for a given network [65]. However prior work focuses on computing metrics that are topological and does not take into account the dynamic behavior that programmable networks are capable of exhibiting. The dynamic behavior of the entire network is determined by a combination of the features of the data-plane and their corresponding control-plane configuration. We propose capturing these dynamics by using simulated models of these networks. The model must be capable of determining, for a given data-plane and configuration for its switches, whether a given pair of hosts remains connected when a set of failure events have occurred. Note that this is in contrast to what we studied in Chapter 3 where our concern was to validate policies that exhaustively validated all possible connectivity and not a given host pair.

We have done two projects where this simulation model has been used in the context of the OpenFlow 1.3 fast-failover mechanism. In each project, we used a simulation model in combination with an appropriate analytic framework to compute a specific metric. In one project, we used variance reduction techniques to compute the expected number of links that will fail in a given network before the controller is called to rescue and restore the paths [66]. In another project we used a simple Monte Carlo method to estimate the probability that a given set of hosts remain connected under a given mobility model for the fast-failover capable wireless switches [67].

The rest of the chapter is organized as follows: Section 4.1 introduces some related work; Section 4.2 describes the motivation of using simulation to compute metrics; Section 4.3 describes the architecture of the proposed data-plane simulator; Section 4.4 describes several design decisions and compares them with our contributions in Chapter 3; Section 4.5 presents the evaluation of our simulator design.

4.1 Related Work

One of the many metrics to characterize the topological resiliency of a network is static reliability. Static reliability refers to the probability that a given set of nodes are all connected when links (not nodes) in the system can fail problematically. It is termed such because it refers to the topological connectivity of the network. Computing this metric for a given network in a closed form is known to be NP-hard [68]. Hence sampling is used. There are previously studied techniques to estimate static reliability using Monte Carlo methods [69]. However simple Monte Carlo methods yield very high relative error when dealing with rare events (e.g. link failures). To that end, sampling techniques using variance reduction approaches have been devised to tackle them [65].

Resilience can also be provisioned by reacting to the failure event once it has occurred. With a centralized control-plane, one responds to failures by waiting for the switches to contact the controller for new routes. However, such a reactive system results in large convergence times as a function of number of flows that are affected by the failures [70]. In this chapter, we are concerned with networks where the fast-failover mechanism has been used.

Theoretically, an emulator such as `mininet` [57] provides the same functionality as a simulator for the programmable network. In practice, however, it does not scale with a large number of switches. Furthermore, using emulation limits the modes of interaction with the model for metrics computation over a large network, and emulation of events such as link failures involves interaction with the Linux kernel. There have been prior efforts to model programmable networks using discrete-event simulation [71], but they did not model the fast failover mechanism.

4.2 Motivation

Resilience metrics provide insights about the robustness of a network to failures and thus inform the choices of network designers. It is often difficult to keep track of individual host-pair connectivity, so a summarizing statistic such as a resilience metric helps capture various properties of the network. The use of metrics also allows the network designers to compare various control-plane programs. The programs ultimately generate flow rules. By analyzing the overall properties of the network that emerge directly from the flow rules, a network designer can choose the control-plane program best suited to her needs.

Furthermore, the rise of programmable data-planes implies that individual data-plane instances (i.e. switches) will behave increasingly more dynamically with minimal control-plane interventions. Such dynamism is especially useful for providing resilience, and the fast failover mechanism is a prime example of such dynamism. However, the use of such dynamic mechanisms also poses a trade-off between network device complexity (and consequently performance challenges) and responsiveness to failures. Shifting more complexity to the data-plane (i.e. to make local forwarding decisions instead of contacting the controller) reduces the time to respond to the random dynamics (i.e. failure of a link or a switch) and consequently leads to lower packet loss, but it also removes key elements of control away from the control-plane. In order to make the appropriate choices in such trade-offs, we need data-plane simulators because the increase in the complexity of data-planes makes them hard to model with the simplifying assumptions used in Chapter 3.

In the rest of this section, we first set the context by describing the various metrics our work concerns directly. We then describe commonly used analytic mechanisms to compute these metrics. Finally, we describe the consequent requirements for a data-plane simulator that it has to meet in order to allow the analytic mechanisms to work.

4.2.1 Examples of Resiliency Metrics

We are primarily concerned with metrics that concern the connectivity of the network in case one or more failure events occur and the fast-failover mechanism is engaged. Some examples of the questions that a designer might ask are:

1. Given a changing topology, what fraction of flows in a given set of flows are connected? This metric is particularly useful to capture the effectiveness of a given flow rule synthesis algorithm in the scenarios involving mobility in wireless settings. This metric can also characterize the impact of cascaded failures on a power grid.

2. How many link or device failures does it take for the controller intervention to be required to enable a given set of flows? This metric is useful to compare the effectiveness of flow rule synthesis algorithms to deal with the impact of a series of bad events.

3. What is the average length of the paths after a set of link or device failures have occurred? This metric is useful to compare synthesis schemes that provide similar resilience guarantees but with different consequences for the eventual performance requirements such as end-to-end delay.

4.2.2 Computing Resiliency Metrics using Active Paths

The resiliency metrics mentioned above can all be computed using Monte Carlo methods. The use of these methods involves multiple simulations of the random phenomenon (i.e. failure events) to compute many samples. While the exact algorithm that lies behind the computation of samples for different metrics may vary, we observe that the computation of the *active* path lies at the heart of computing each of the examples presented above.

In the context of a dynamic data-plane that supports fast-failover mechanism, an active path is defined as the sequence of ports between a specific pair of hosts which the packets traverse *after* a set of links in the network have failed. The computation of an active paths requires the simulation of the data-plane along with its configuration and a description of the incident failure events. The metrics mentioned previously can be sampled by using the fact of whether an active path exists or not. In some cases, in case of its existence, the length of the path (in number of ports/switch-hops) can also be used to constitute the samples.

Finally, in order to come up with an estimate for the metric, the collected samples undergo output processing. This process also yields confidence intervals. Since the computation of a network-wide resiliency metric may involve simulating rare events, it may take a very large number of samples to estimate the metric with high confidence. This results in large computation times. Hence, variance reduction techniques have been proposed to achieve high confidence with fewer samples. However, variance reduction techniques require more state from the simulated network and not just the final sample output. We discuss this and other requirements of the simulation architecture next.

4.2.3 Requirements for a Data-Plane Simulator Architecture

Based on the nature of resiliency metrics and mechanisms that are used to compute them, we summarize the key requirements for the data-plane simulator architecture below:

- **Fidelity:** The simulator needs to capture the relevant behaviors of a switch with very high fidelity based on three factors: The OpenFlow specification, the configuration inside the switch and finally the contents of the header of the packets arriving at an ingress port. Such behaviors necessarily include the packet processing pipeline, the simple output and fast-failover mechanisms as described in the Chapter 2.

  Beyond a single switch, in order to compute the active paths, the network-wide behaviors must be captured as well. To that end, the

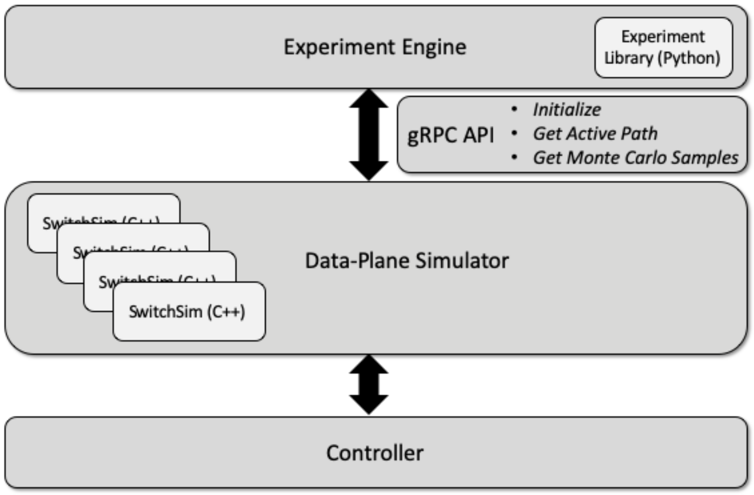


Figure 4.1: Data-plane simulator architecture

simulator needs to track the modifications applied to a packet as well as the links it traverses.

- **Performance:** Unlike other analysis which compute all possible actions for all possible flows, in the data-plane simulator we are focused on specifically identified host-pairs and the active paths between them.

- **Scalability:** The simulator must tackle topology sizes that are typical in CI networks without having to resort to distributed computation.

- **Convenient API:** The simulator must expose the API for repeating experiments. It must make it easy and efficient to collect the data required to generate statistics.

- **Flexibility:** The simulator must have a flexible design that allows computation of a variety of metrics around the active path computation. Its API must also support reporting of the corresponding state with the sample to facilitate various variance reduction mechanisms.

4.3 The Data-Plane Simulator Architecture

The primary input of the data-plane simulator is a snapshot of the logically centralized *state* of the programmable CI network of interest. The snapshot consists of the network topology and the forwarding rules installed on switches. The controller in a programmable network is the purveyor of this state and makes it available via a *northbound* API to be used by the applications. As illustrated in Figure 4.1, our simulator uses the northbound API to access a current snapshot of the SDN state. Note that, by using a snapshot of the state, our architecture allows the metrics computation to be performed offline so that it does not affect the operation of the network itself. In the remainder of this section, we provide more details on how the proposed simulator architecture meets the requirements mentioned previously.

In order to provide flexibility for computing various types of metrics, we propose a bifurcated architecture: a compiled network model which exposes a language-independent API to set up experiments, change conditions, and make calls upon the network model. The data-plane simulator is implemented in C++. The simulator implements a gRPC server that exposes the API for interacting with it [72]. We also wrote a library of experiments in Python that encapsulate a gRPC client that implements the computation of samples for the metrics described above. Both components of the current implementation are language independent. However, the choice of C++ and Python for these two tasks is primarily driven by the performance consideration and the ecosystem of libraries available for each of them respectively.

In order to provide fidelity, the architecture explicitly models the OpenFlow specification in a module called `SwitchSim`. As illustrated in Figure 4.1, this module lies at the heart of the simulator and closely mimics the forwarding actions and transformations made on a packet passing through it. To understand the importance of this element of the design it is important to know that the OpenFlow specification supports more than just simple or failover routing. The switch might block a packet entirely. It might change IP and port numbers in a packet's header to implement network address translation (NAT). It might modify other bits in the packet header. The point is that the data-plane simulator is designed to be used in an operational environment with potentially complex switch configurations, and so `SwitchSim` is an executable implementations of the OpenFlow specification.

Our proposed architecture makes several explicit choices to address performance concerns. Multi-threading is used for performance gains. Each Monte Carlo sample is computed in its own thread. As detailed above, each sample is computed by obtaining active paths for a set of host pairs. Hence, due to multi-threading, computations across different samples run in parallel. However, in order to save memory and hence scale to large topologies, we make separate copies of only the state that is different for each thread. For example, each thread may be dealing with a different set of failed links due to randomness, but the network's configuration snapshot containing the topology and flow rules stays the same across all threads. Furthermore, we observed that each sample computation thread only reads the shared state. Hence, we initialize this shared state before starting the threads so their reads can proceed without any locks.

Finally, gRPC API is absolutely crucial for data collection for metrics computation. The API is structured but easily extensible and can be designed to provide state beyond the sample outputs. For example, we used it even to report the time it takes to compute samples as reported in the next section.

4.4 Design

The design of the data-plane simulator borrows several ideas from that of the FlowValidator described in Section 3.5. It borrows the key idea that for the purpose of forwarding packets, the meaningful decisions occur only at the packet processing pipeline (i.e. flow tables in OpenFlow). The ports at switches are conduits that are followed from one table to another in the network, until a dead-end is found due to lack of flow rules or failure of links or a host is reached.

However, we modeled the effects of set of packets interacting with flow rules using header spaces in Chapter 3. In the data-plane simulator we avoid that in favor of explicit representation of each flow table and flow rule. The algorithm to determine whether an active path between two hosts exists is essentially a specialized breadth-first search across the topology. The search carries a list of links that have presumably failed and avoids taking those links. The search also checks if the flow rules have configured a fast-failover path and follows those paths. Thus, the size of the topology affects the runtime of the algorithm linearly. The simulator design allows it to scale to a very high number of switches. The network topology and configuration also affect the memory that the simulator consumes linearly.

The search does not use a model that represents aggregates of packet headers. It only models the contents of a single packet. Hence, while finding an active path, instead of propagating a chunk of the header space, the search only propagates the contents of a single packet header. This header models the flow of interest and is compared explicitly against the match of the flow rules encountered in the tables, instead of finding intersections of header space. Hence, the algorithm used in this search is essentially the same as Algorithm 1. However, the key difference is that the $ATS_{p,d}$ here represents a single packet header. Naturally, this design choice results in constraints on the Policy Validation queries that can be made. In particular, now a single policy query can only concern a specific set of values in a header, whereas using Flow Validator the fate of an entire set of packets can be determined using a single query. However, in practice, this constraint is not limiting for the Monte Carlo sample computations described in Section 4.2.

4.5 Evaluation

We present an empirical evaluation of the performance of the data-plane simulator. The objective of the simulator is to compute Monte Carlo samples to study the effectiveness of a flow rule synthesis algorithm that uses fast-failover mechanism. Each Monte Carlo sample represents the fraction of flows that remain connected even if a given sequence of link failures occur were to occur.

We performed this evaluation to test the scalability of the simulator for an increasing number of switches. We assumed that each switch has a single source or destination host attached to it. Both the source and destination hosts and the existence of links among the switches are determined randomly in each topology. We set the total number of flows in our experiments to 32, regardless of the size of the topology. Each flow has been synthesized to have a path in the topology that can sustain a single link failure by using the fast-failover mechanism in the OpenFlow. These random topologies are motivated by our previous work [67].

We computed 100 samples on these topologies and measured the time it takes to perform different operations. We repeated the experiment 20 times and show the average time it took to perform various operations in Figure 4.2. There is the Initialization time which indicates how long it took on average

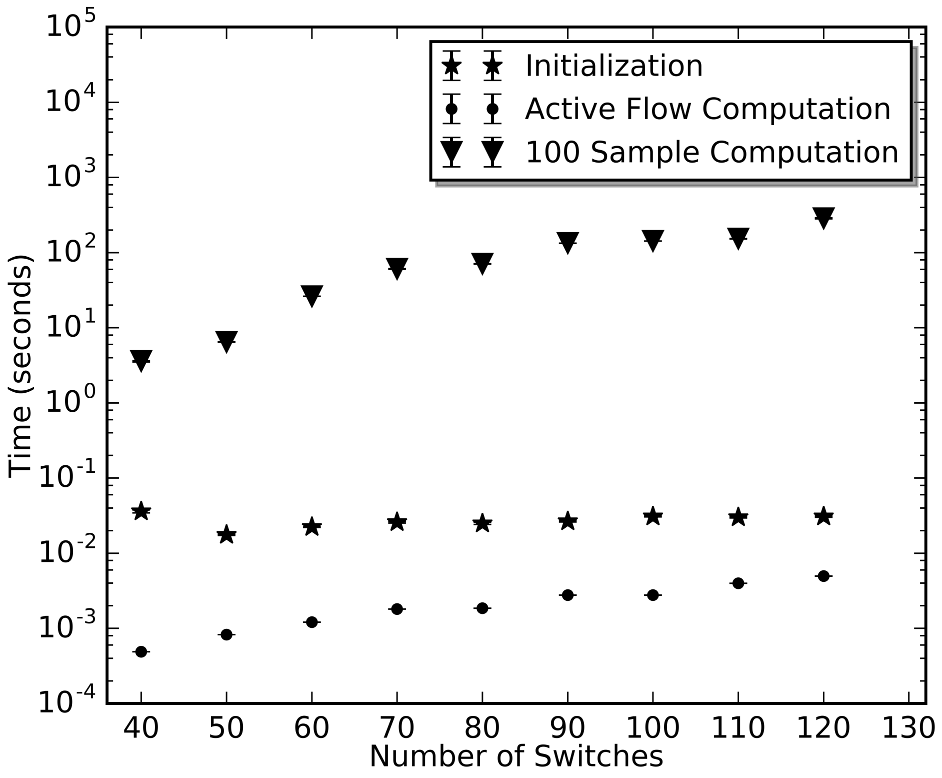


Figure 4.2: Time taken by the data-plane simulator to initialize a new topology, determine the path of an active flow and compute 100 samples. Each point represents an average over 20 iterations.

to set up the experiment with a completely new topology *before* any other computations occur. We found this time to be independent of the size of the topology. Then, we measured the time it takes to determine whether an active path (i.e. taking into account the effect of the fast-failover mechanism) exists between a randomly picked pair of hosts in the topology. Evidently, this time increases with the size of the topology. Finally, we measured the total time it takes to compute 100 samples. This time is simply a multiple of the time it takes to compute a single active path and thus it also increases as the size of the topology increases.

4.6 Case Study

In order to demonstrate the effectiveness of the data-plane simulator, we performed a case study to see how the use of fast-failover mechanism affects the nature of connectivity in a random topology when a set of links fail arbitrarily.

The setup topology had a topology with 40 switches and 186 links between randomly sampled pairs between them. We changed the number of synthesized flows in the same underlying topology. Each switch had a single host attached to it, which can serve as a source or destination for a unidirectional flow. Each flow has been synthesized to use the fast-failover mechanism and can sustain the failure of a single link. The flow paths in the random topology were chosen using the Dijkstra's shortest path algorithm. For providing the failover for each link, we simply assumed that the particular link did not exist in an alternative topology and found the shortest path in the same. We then tried to randomly sample an increasing number of links and failed them. In particular, we failed 5, 10, 15 and 20 links in the topology and checked how many of the originally synthesized flows were still active with the our of data-plane simulator.

We repeated the experiment 100 times and Figure 4.3 presents our results. The y-axis is the fraction of synthesized flows that remained active even after the specific number of links have failed. The first major takeaway from the experiment is that regardless of the number of link failures, as the number of flows increases, a greater fraction of them become inactive due to link failures. Secondly, the number of failed links matters, but only to a certain point. Notice that both 15 and 20 link failure fractions are essentially overlapping. However, there is a wide gap between the effects of 5 and 20 failed links.

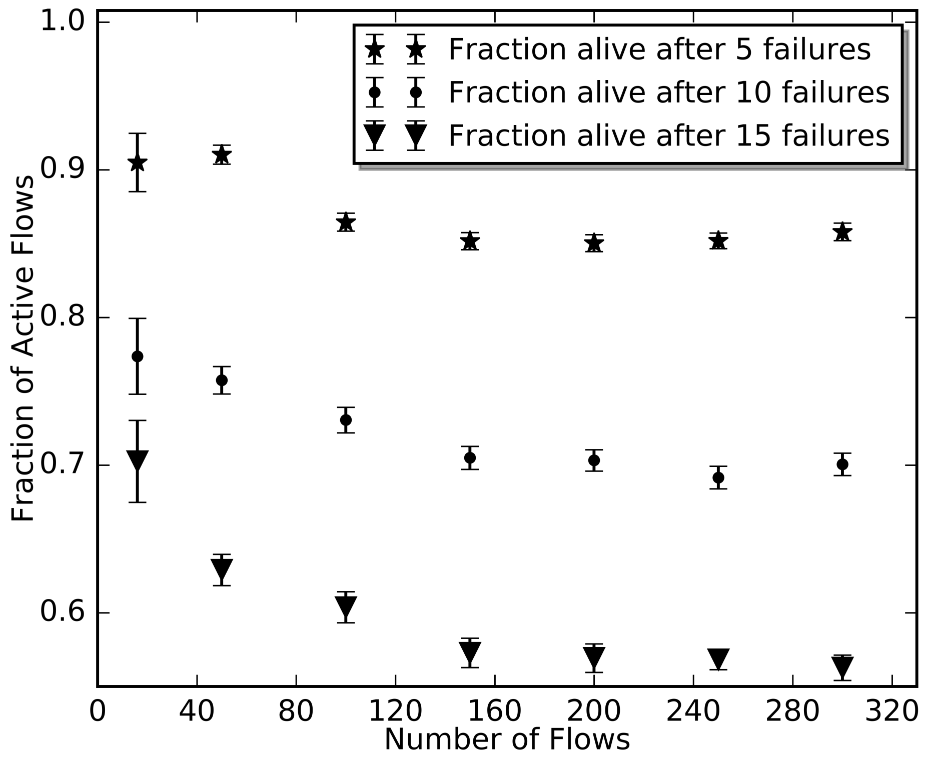


Figure 4.3: Case study: Fraction of flows that remain active when an increasingly larger set of random sampled links have failed.

CHAPTER 5

SYNTHESIS: BANDWIDTH AND DELAY GUARANTEES

Many components in CI systems need to generate flows that require hard real-time guarantees [73]. These guarantees are for both bandwidth and maximum end-to-end delay experienced by the critical flows. Examples of such systems include avionics, automobiles, industrial control systems, power substations, manufacturing plants, etc. Current CI systems often have separate networks (hardware and software) for different critical flows. This leads to significant overheads (equipment, management, weight, etc.) and also potential for errors/faults and even increased attack surfaces and vectors. Existing systems, e.g., avionics full-duplex switched Ethernet (AFDX) [74, 75, 76], controller area network (CAN) [77], etc., that are in use in many of these domains are proprietary, complex, expensive and require custom hardware.

As described in Chapter 1, one of the advantages of using programmable networks is the mechanism to push down rules to the commercial, off-the-shelf (COTS) switches that can, to a fine level of precision, manage the bandwidth assigned to a flow through the entire network. However, there is no direct mechanism to guarantee a certain end-to-end delay for any given flow. In this chapter we present mechanisms to *guarantee end-to-end delays for critical flows on networks constructed using programmable switches.* The main contributions of this work are summarized as follows:

1. We demonstrate the need for isolating flows into separate queues to provide stable end-to-end delays (Section 5.1) even in the presence of other types of traffic in the system.

2. We present mechanisms to guarantee delay constraints for individual end-to-end flows in hard real-time systems based on COTS hardware (Sections 5.3, 5.4 and 5.5) by formulating a multi-constraint problem.

We empirically evaluate the effectiveness of the proposed approach with various topologies and UDP traffic (Section 5.7) on a widely-used emulation platform. Our results demonstrate that the end-to-end delay experienced by the critical flows falls within their user specified timing deadline.

5.1 Motivating Experiment

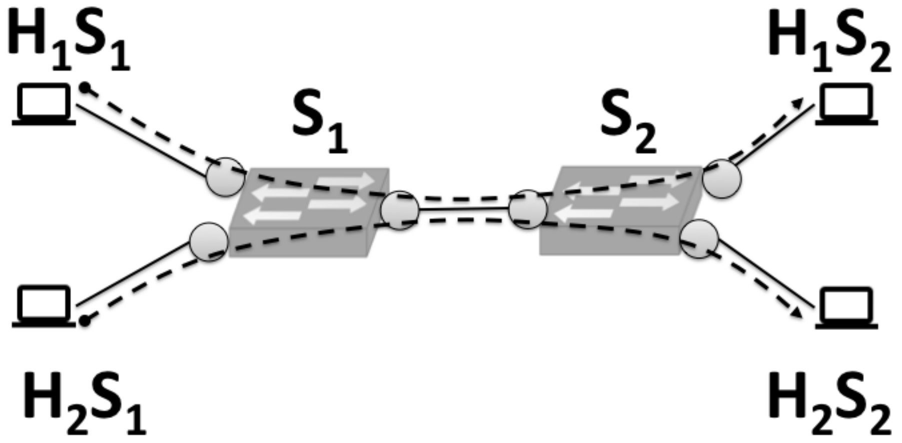


Figure 5.1: The two-switch, four-host topology used in the experiments with the active flows.

We intend to synthesize configurations for critical traffic such that it ensures *complete isolation of packets for each designated critical flow at each switch in its path.*

In order to test how using output queues can provide isolation to flows in a network so that each can meet its delay and bandwidth requirements simultaneously, we performed experiments using mininet. The experiments use a simple topology that contains two switches (`s1`, `s2`) connected via a single link as shown in Figure 5.1. Each switch has two hosts connected to it.

We configured flow rules and queues in the switches to enable connectivity among hosts at one switch with the hosts at other switch. We experimented

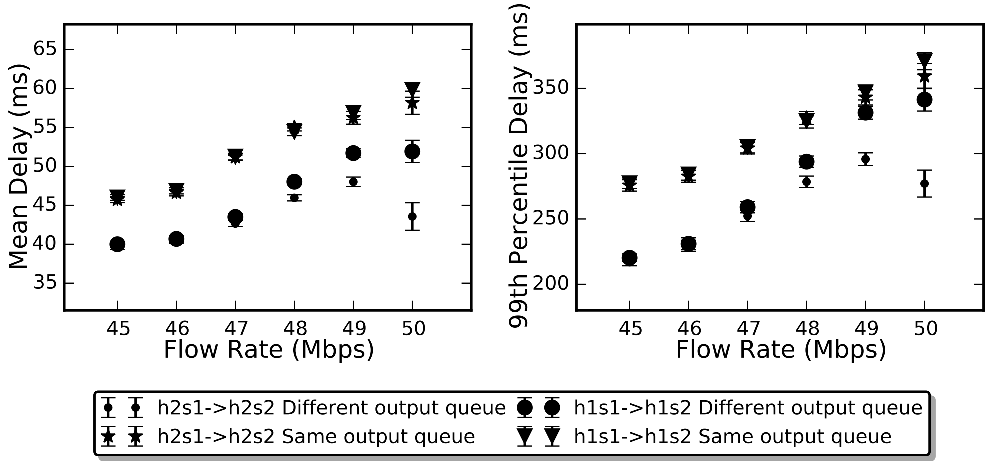


Figure 5.2: The measured mean and 99^{th} percentile per-packet delay for the packets in the active flows in 30 iterations.

with two ways to queue the packets as they cross the switch-to-switch link: *(i)* in one case, we queue packets belonging to the two flows *separately* in two queues (i.e., each flow gets its own queue), each configured at a maximum rate of 50 Mbps *(ii)* in the second case, we queue packets from both flows in the *same queue* configured at a maximum rate of 100 Mbps.

After configuring the flow rules and queues, we used `netperf` [78] to generate the following packet flows: the first starting at the host `h1s1` destined to host `h1s2` and the second starting at host `h2s1` with a destination host `h2s2`. Both flows are identical and are triggered simultaneously to last for 15 seconds. We changed the rate at which the traffic is sent across both flows to measure the average per-packet delay. Figure 5.2 plots the average value and standard error over 30 iterations. The x-axis indicates the rate at which the traffic is sent via `netperf`, while the y-axis shows the average per-packet delay. The following key observations stand out:

1. The per-packet average delay increases in both cases as the traffic send rate approaches the configured rate of 50 Mbps. This is an expected queue-theoretic outcome and motivates the need for slack allocations for all applications in general. For example, if an application requires a bandwidth guarantee of 1 Mbps, it should be allocated 1.1 Mbps for minimizing jitter.

2. The case with separate queues experiences lower average per-packet de-

lay when flow rates approach the maximum rates. This indicates that when more than one flow uses the same queue, there is interference caused by both flows with each other. This becomes a source of unpredictability and eventually may cause the end-to-end delay guarantees for the flows to be not met or perturbed significantly.

Thus, isolating flows using separate queues results in lower and more stable delays, especially when traffic rate in the flow approaches the configured maximum rates. Such isolation leads to a correct-by-design approach that ensures that each flow is allocated bandwidth at each switch such that it does not experience queueing delays. The maximum processing delay along a single link can be measured and used as input to a path allocation algorithm that we describe in the following section.

5.2 Related Work

There have been several efforts to study the provisioning a network such that it meets bandwidth and/or delay constraints for the traffic flows. Results from the network calculus [79] framework offer a concrete way to model the various abstract entities and their properties in a computer network. NC-based models, on the other hand, do not prescribe any formulation of flows that meet given delay and bandwidth guarantees. For synthesis, the NP-complete MCP comes close and Shingang et al. formulated a heuristic algorithm [80] for solving MCP. We model our delay and bandwidth constraints based on their approach.

There are recent standardization efforts such as IEEE 802.11Qbv [81] which aim to codify best practices for provisioning QoS using Ethernet. These approaches focus entirely on meeting guarantees and do not attempt to optimize link bandwidth. However, the global view of the network provided by the SDN architecture allows us to optimize path layouts by formulating the problem as an MCP problem.

There are prior attempts at provisioning SDN with worst-case delay and bandwidth guarantees. Azodolmolky et al. proposed a network calculus based model [82] for a single SDN switch that provides an upper bound on delays experienced by packets as they cross through the switch. Guck et al. used mixed integer program (MIP) based formulation [83] for provisioning end-to-end flows with delay guarantees; however, their approach does not optimize the bandwidth allocation to each queue used by the end-to-end flows at an individual switch. There are approaches [84, 85] that have used queues to rate-limit network traffic and improve end-to-end delay for cloud applications (*e.g.,* MapReduce). However, they do not try to meet a specific end-to-end delay deadline for a given flow; rather, they accumulate all traffic belonging to a given tenant VM and apply the queue constraints on the host level.

Avionics full-duplex switched Ethernet (AFDX) [74, 75, 76] is a deterministic data network developed by Airbus for safety critical applications. The switches in AFDX architecture are interconnected using full duplex links, and static paths with predefined flows that pass through network are set up.

Though such solutions aim to provide deterministic QoS guarantees through static routing, reservation and isolation, they impose several limitations on optimizing the path layouts and on different traffic flows. There have been studies toward evaluating the upper bound on the end-to-end delays in AFDX networks [76]. The evaluation seems to depend on the AFDX parameters, though.

Furthermore, there are several protocols proposed in the automotive communication networks such as controller area network (CAN) [77] and FlexRay [86]. While these protocols are designed to provide delay guarantees, they are limited in their applicability to networks with traffic flows with different bandwidth requirements and complex network topologies. The SDN architecture is useful in extending the guarantees in such scenarios. Thus, in this chapter, we propose a flexible framework to configure COTS components and meet end-to-end delay and bandwidth guarantees with optimized path layouts.

5.3 System Model

Consider a network topology (N) with open flow switches and controller, and a set of real-time flows (F) with specified delay and bandwidth guarantee requirements. *The problem is to find paths for the flows (through the topology) such that the flow requirements (i.e., end-to-end delays) can be guaranteed for the maximum number of critical flows.* We model the network as an undirected graph $N(V, E)$ where V is the set of nodes, each representing a switch port in a given network and E is set of the edges, each representing a possible path for packets to go from one switch port to another. Each port $v \in V$ has a set of queues v_q associated with it, where each queue is assigned a fraction of bandwidth on the edge connected to that port.

Consider a set F of unidirectional, real-time flows that require delay and bandwidth guarantees. The flow $f_k \in F$ is given by a four-tuple (s_k, t_k, D_k, B_k), where $s_k \in V$ and $t_k \in V$ are ports (the source and destination respectively) in the graph, D_k is the maximum delay that the flow can tolerate and B_k is the maximum required bandwidth by the flow. We assume that flow priorities are distinct and the flows are prioritized based on a *delay-monotonic* scheme, viz., the end-to-end delay budget represents higher priority (i.e., $pri(f_i) > pri(f_j)$ if $D_i < D_j$, $\forall f_i, f_j \in F$ where $pri(f_k)$ represents priority of f_k).

For a flow to go from the source port s_k to a destination port t_k, it needs to traverse a sequence of edges, i.e., a flow path $\mathcal{P}_k$. The problem, then, is to synthesize flow rules that use queues at each edge $(u, v) \in \mathcal{P}_k$ that can handle *all* flows F in the given system while still meeting each flow's requirement. If $d_{f_k}(u, v)$ and $b_{f_k}(u, v)$ are the worst-case delay faced by the flow and bandwidth assigned to the flow at each edge $(u, v) \in E$ respectively, then $\forall f_k \in F$ and $\forall (u, v) \in \mathcal{P}_k$, and the following constraints need to be satisfied:

$$\sum_{(u,v) \in \mathcal{P}_k} d_{f_k}(u, v) \leq D_k, \quad \forall f_k \in F \tag{5.1}$$

$$b_{f_k}(u, v) \geq B_k, \quad \forall (u, v) \in \mathcal{P}_k, \forall f_k \in F. \tag{5.2}$$

This problem needs to be solved at two levels:

- *Level 1*: Finding the path layout for each flow such that it satisfies the flows' delay and bandwidth constraints. We formulate this problem as a multi-constrained path (MCP) problem and describe the solution in Sections 5.4 and 5.5.

- *Level 2*: Mapping the path layouts from Level 1 on to the network topology by using the mechanisms available in OpenFlow. We describe details of our approach in Section 5.6.

In addition to the aforementioned delay and bandwidth constraints (see Eqs. (5.1) and (5.2)), we need to map flows assigned to a port to the queues at the actual ports. Two possible approaches are: *(a) allocate each flow to an individual queue* or *(b) multiplex flows onto a smaller set of queues* and dispatch the packets based on priority. In fact, as we illustrate in the following section, the queuing approach used will impact the delays faced by the flows at each link. Our intuition is that the *end-to-end delays are lower and more stable* when *separate queues* are provided to each critical flow – especially as the rates for the flows get closer to their maximum assigned rates. Given the nature of many CI networks, the number of critical flows is often limited and well defined (e.g., known at design time). Hence, such overprovisioning is an acceptable design choice – from computing power to network resources (for instance, one queue per critical real time flow).

5.4 Path Layout: Overview and Solution

We now present a more detailed version of the problem (composing paths that meet end-to-end delay constraints for critical real-time flows) and also an overview of our solution. First, we briefly describe the classical multi-constrained path problem in the next sub-section. The remainder of this section describes our contribution of formulating the path layout problem as an instance of the MCP problem.

5.4.1 The Multi-constrained Path (MCP) Problem

Assume a directed graph $G(V, E)$, a source vertex v_1 and a destination vertex v_k in the set E. Assume also that each arc in the set E has two non-negative weights specified by the weight functions: $w_1 \rightarrow R_0^+$ and $w_2 \rightarrow R_0^+$.

A multi-constrained path p is a path $(v_1 \rightarrow v_2 \rightarrow v_3 \cdots \rightarrow v_k)$ such that it meets following two constraints ($C_1 \in R_0^+$ and $C_2 \in R_0^+$) simultaneously:

$$W_1(p) = \sum_{i=1}^{k-1} w_1(v_i, v_{i+1}) \leq C_1. \tag{5.3}$$

$$W_2(p) = \sum_{i=1}^{k-1} w_2(v_i, v_{i+1}) \leq C_2. \tag{5.4}$$

The problem $MCP(G, v_1, v_k, w_1, w_2, C_1, C_2)$ is known to be NP-complete. [87].

5.4.2 Problem Overview

Let $\mathcal{P}_k$ be the path from s_k to t_k for flow f_k that needs to be determined. Let $\mathfrak{D}(u, v)$ be the delay incurred on the edge $(u, v) \in E$.

The total delay for f_k over the path $\mathcal{P}_k$ is given by c. Therefore we define the constraint on end-to-end delay for the flow f_k as:

$$\mathfrak{D}_k(\mathcal{P}_k) \leq D_k. \tag{5.5}$$

Note that the end-to-end delay for a flow over a path has the following delay components: *(a)* processing time of a packet at a switch, *(b)* propagation on the physical link, *(c)* transmission of packet over a physical link, and *(d)* queuing at the ingress/egress port of a switch. As discussed in Section 5.3, we use separate queues for each flow with assigned required rates. We also overprovision the bandwidth for such flows so that critical real-time flows do not experience queueing delays. Hence, we consider queuing delays to be negligible. The other components of delay can be empirically estimated.

The second constraint that we consider in this work is *bandwidth utilization*, that for an edge (u, v) for a flow f_k, can be defined as:

$$\mathfrak{B}_k(u, v) = \frac{B_k}{B_e(u, v)} \tag{5.6}$$

where B_k is the bandwidth requirement of f_k and $B_e(u, v)$ is *residual* (viz., available) bandwidth of an edge $(u, v) \in E$. For the purposes of establishing a constraint for the setup of the MCP, bandwidth utilization over a path

$(\mathcal{P}_k)$, for a flow f_k is defined as:

$$\mathfrak{B}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \mathfrak{B}_k(u,v). \tag{5.7}$$

Hence, note that the bandwidth utilization over a path $\mathcal{P}_k$ for flow f_k is bounded by

$$\mathfrak{B}_k(\mathcal{P}_k) \leq \max_{(u,v)\in E} \mathfrak{B}_k(u,v)|V|. \tag{5.8}$$

where $|V|$ is the cardinality of a set of nodes (ports) in the topology N. Therefore in order to ensure that the bandwidth requirement B_k of the flow f_k is guaranteed, it suffices to consider the following constraint on bandwidth utilization

$$\mathfrak{B}_k(\mathcal{P}_k) \leq \widehat{B}_k \tag{5.9}$$

where $\widehat{B}_k = \max\limits_{(u,v)\in E} \mathfrak{B}_k(u,v)|V|$. Note that the constraint in Eq. (5.9) is loose. It can be tightened to reflect the occupancy of the flow along its path. However, the path is not known a priori. Furthermore, the cardinality $|V|$ can also be replaced with the diameter of the topology. While such replacement will make the constraint tighter, it remains to explore how it will affect the utilization.

Remark 1 *The selection of an optimal path for each flow $f_k \in F$ subject to delay and bandwidth constraints in Eq. (5.5) and (5.9), respectively, can be formalized as an MCP problem.*

In order to solve the NP-complete problem, we use a polynomial-time heuristic proposed by Chen and Nahrstedt [80]. The next subsection describes our approach to use this heuristic.

5.4.3 Polynomial-time Solution to the Path Layout Problem

The key idea behind the heuristic presented by Chen and Nahrstedt [80] is to *relax* one of the constraints (in our case delay or bandwidth) at a time and try to obtain a solution. If the original MCP problem has a solution, one of the relaxed versions of the problem will also have a solution [80]. However, in order use the polynomial-time heuristic, we first define following quantities pertaining to the constraints in our context:

$$\widetilde{\mathfrak{D}}_k(u,v) = \left\lceil \frac{X_k \cdot \mathfrak{D}(u,v)}{D_k} \right\rceil \tag{5.10}$$

$$\widetilde{\mathfrak{B}}_k(u,v) = \left\lceil \frac{X_k \cdot \mathfrak{B}_k(u,v)}{\widehat{B}_k} \right\rceil \tag{5.11}$$

where X_k is a given positive integer. For instance, if we relax the bandwidth constraint (e.g., represent $\mathfrak{B}_k(\mathcal{P}_k)$ in terms of $\widetilde{\mathfrak{B}}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \widetilde{\mathfrak{B}}_k(u,v)$), Eq. (5.9) can be rewritten as

$$\widetilde{\mathfrak{B}}_k(\mathcal{P}_k) \leq X_k. \tag{5.12}$$

The solution to this relaxed problem will also be a solution to the original MCP [80]. Likewise, if we relax the delay constraint, Eq. (5.5) can be rewritten as

$$\widetilde{\mathfrak{D}}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \widetilde{\mathfrak{D}}_k(u,v) \leq X_k. \tag{5.13}$$

Let the variable $d_k[v,i]$ preserve an *estimate* of the path from s_k to t_k for $\forall v \in V$, $i \in \mathbb{Z}^+$ (refer to Algorithm 4). There exists a solution (*e.g.,* a path $\mathcal{P}_k$ from s_k to t_k) if *any* of the two conditions is satisfied when the *original MCP problem is solved by the heuristic.*

- *When the bandwidth constraint is relaxed:* The delay and (relaxed) bandwidth constraints, i.e., $\mathfrak{D}_k(\mathcal{P}_k) \leq D_k$ and $\widetilde{\mathfrak{B}}_k(\mathcal{P}_k) \leq X_k$ are satisfied if and only if

$$d_k[t,i] \leq D_k, \quad \exists i \in [0, X_k] \wedge i \in \mathbb{Z}.$$

- *When the delay constraint is relaxed:* The (relaxed) delay and bandwidth constraints, i.e., $\widetilde{\mathfrak{D}}_k(\mathcal{P}_k) = \sum_{(u,v)\in\mathcal{P}_k} \widetilde{\mathfrak{D}}_k(u,v) \leq X_k$ and $\mathfrak{B}_k(\mathcal{P}_k) \leq \widehat{B}_k$ are satisfied if and only if

$$d_k[t,i] \leq X_k, \quad \exists i \in [0, \widehat{B}_k] \wedge i \in \mathbb{Z}.$$

5.5 Algorithm Development

The proposed heuristic solution of the MCP problem, as summarized in Algorithm 4, works as follows. Let

$$\Delta(v, i) = \min_{\mathcal{P} \in P(v,i)} W_1(\mathcal{P}) \tag{5.14}$$

where $P(v, i) = \{\mathcal{P} \mid W_2(\mathcal{P}) = i, \mathcal{P}$ is any path from s to $t\}$ is the smallest $W_1(\mathcal{P})$ of those paths from s to v for which $W_2(\mathcal{P}) = C_2$. For each node $v \in V$ and each integer $i \in [0, \cdots, C_2]$ we maintain a variable $d[v, i]$ that keeps an estimation of the smallest $W_1(\mathcal{P})$. The variable initialized to $+\infty$ (Line 3), which is always greater than or equal to $\delta(v, i)$. As the algorithm executes, it makes better estimation and eventually reaches $\Delta(v, i)$ (Line 8-15). Lines 3-17 in Algorithm 4 are similar to the single-cost path selection approach presented in earlier work [80, Sec. 2.2], and for the purposes of this work, we have extended the previous approach for our formulation.

We store the path in the variable $\pi[v, i], \forall v \in V, \forall i \in [0, \cdots, C_2]$. When the algorithm finishes the search for path (Line 17), there will be a solution if and only if the following condition is satisfied [80]:

$$\exists i \in [0, \cdots, C_2], \quad d[t, i] \leq C_1. \tag{5.15}$$

If it is not possible to find any path (e.g., the condition in Eq. (5.15) is not satisfied), the algorithm returns False (Line 41). If there exists a solution (Line 19), we extract the path by backtracking (Line 21-29). Notice that the variable $\pi[v, i]$ keeps the immediate preceding node of v on the path (Line 13). Therefore, the path can be recovered by tracking π starting from destination t through all immediate nodes until reaching the source s. Based on this MCP abstraction, we developed a path selection scheme considering delay and bandwidth constraints (Algorithm 5) that works as follows.

5.5.1 Path Layout

Let us consider `MCP_HEURISTIC`$(N, s, t, W_1, W_2, C_1, C_2)$, an instance of polynomial-time heuristic solution to the MCP problem that finds a path $\mathcal{P}$ from s to t in any network N, satisfying constraints $W_1(\mathcal{P}) \leq C_1$ and $W_2(\mathcal{P}) \leq C_2$.

Algorithm 4 Multi-constraint Path Selection

Input: The network $N(V, E)$, source s, destination t, constraints on links $W_1 = [w_1(u,v)]_{\forall(u,v)\in E}$ and $W_2 = [w_2(u,v)]_{\forall(u,v)\in E}$, and the bounds on the constraints $C_1 \in \mathbb{R}^+$ and $C_2 \in \mathbb{R}^+$ for the path from s to t.

Output: The path $\mathcal{P}^*$ if there exists a solution (*e.g.,* $W_1(\mathcal{P}^*) \leq C_1$ and $W_2(\mathcal{P}^*) \leq C_2$), or False otherwise.

```
 1: function MCP_HEURISTIC($N, s, t, W_1, W_2, C_1, C_2$)
 2:     /* Initialize local variables */
 3:     $d[v,i] := \infty, \pi[v,i] := NULL, \quad \forall v \in V, \ \forall i \in [0, C_2] \land i \in \mathbb{Z}$
 4:     $d[s,i] := 0 \ \ \forall i \in [0, C_2] \land i \in \mathbb{Z}$
 5:     /* Estimate path */
 6:     for $i \in |V| - 1$ do
 7:         for each $j \in [0, C_2] \land j \in \mathbb{Z}$ do
 8:             for each edge $(u,v) \in E$ do
 9:                 $j' := j + w_2(u,v)$
10:                 if $j' \leq C_2$ and $d[v,j'] > d[u,j] + w_1(u,v)$ then
11:                     /* Update estimation */
12:                     $d[v,j'] := d[u,j] + w_1(u,v)$
13:                     $\pi[v,j'] := u$ /* Store the possible path */
14:                 end if
15:             end for
16:         end for
17:     end for
18:     /* Check for solution */
19:     if $d[t,i] \leq C_1$ for $\exists i \in [0, C_2] \land i \in \mathbb{Z}$ then
20:         /* Solution found, obtain the path by backtracking */
21:         $\mathcal{P} := \emptyset$, $done :=$ False, $currentNode := t$
22:         /* Find the path from t to s */
23:         while not $done$ do
24:             for each $j \in [0, C_2] \land j \in \mathbb{Z}$ do
25:                 if $\pi[currentNode, j]$ not $NULL$ then
26:                     add $currentNode$ to $\mathcal{P}$
27:                     if $currentNode = s$ then
28:                         $done :=$ True /* Backtracking complete */
29:                         break
30:                     end if
31:                     /* Search for preceding hop */
32:                     $currentNode := \pi[currentNode, j]$
33:                     break
34:                 end if
35:             end for
36:         end while
37:         /* Reverse the list to obtain a path from s to t */
38:         $\mathcal{P}^* :=$ reverse($\mathcal{P}$)
39:         return $\mathcal{P}^*$
40:     else
41:         return False /* No Path found that satisfies $C_1$ and $C_2$ */
42:     end if
43: end function
```

```
Algorithm 5 Layout Path Considering Delay and Bandwidth Constraints
Input: The network $N(V,E)$, set of flows $F$, delay and bandwidth utilization constraints on links $\mathfrak{D}_k = [\mathfrak{D}_k(u,v)]_{\forall (u,v)\in E}$, $\widetilde{\mathfrak{D}}_k = [\widetilde{\mathfrak{D}}_k(u,v)]_{\forall (u,v)\in E}$ and $\mathfrak{B}_k = [\mathfrak{B}_k(u,v)]_{\forall (u,v)\in E}$, $\widetilde{\mathfrak{B}}_k = [\widetilde{\mathfrak{B}}_k(u,v)]_{\forall (u,v)\in E}$, for each flow $f_k \in F$, respectively, and the delay and bandwidth bounds $D_k \in \mathbb{R}^+$ and $\widehat{B}_k \in \mathbb{R}^+$, respectively, and positive constant $X_k \in \mathbb{Z}$, $\forall f_k \in F$.
Output: The path vector $\boldsymbol{\mathcal{P}} = [\mathcal{P}_k]_{\forall f_k \in F}$ where $\mathcal{P}_k$ is the path if the delay and bandwidth constraints (e.g., $\mathfrak{D}_k(\mathcal{P}_k) \le D_k$ and $\mathfrak{B}_k(\mathcal{P}_k) \le \widehat{B}_k$) are satisfied for $f_k$, or False otherwise.
 1: for each $f_k \in F$ (starting from higher to lower priority) do
 2:     Discard the the links for which $B_{e'}(u,v) < B_k, \forall e' \in E$
 3:     /* Relax bandwidth constraint and solve */
 4:     Solve MCP_HEURISTIC($N, s_k, t_k, \mathfrak{D}_k, \widetilde{\mathfrak{B}}_k, D_k, X_k$) by using Algorithm 4
 5:     if SolutionFound then   /* Path found for $f_k$ */
 6:         /* Add path to the path vector $\boldsymbol{\mathcal{P}}$ */
 7:         $\mathcal{P}_k := \mathcal{P}^*$ where $\mathcal{P}^*$ is the solution obtained by Algorithm 4
 8:     else
 9:         /* Relax delay constraint and try to obtain the path */
10:         Solve MCP_HEURISTIC($N, s_k, t_k, \widetilde{\mathfrak{D}}_k, \mathfrak{B}_k, X_k, \widehat{B}_k$) by using Algorithm 4
11:         if SolutionFound then
12:             /* Path found by relaxing delay constraint */
13:             $\mathcal{P}_k := \mathcal{P}^*$ /* Add path to the path vector */
14:             /* Update remaining available bandwidth */
15:             $B_e(u,v) := B_e(u,v) - B_k,\ \forall (u,v) \in \mathcal{P}_k$
16:         else
17:             $\mathcal{P}_k :=$ False   /* Unable to find any path for $f_k$ */
18:         end if
19:     end if
20: end for
```

For each flow $f_k \in F$, starting with highest (e.g., the flow with tighter delay requirement) to lowest priority, we first keep the delay constraint unmodified and relax the bandwidth constraint by using Eq. (5.11) and solve MCP_HEURISTIC$(N, s_k, t_k, \mathfrak{D}_k, \widetilde{\mathfrak{B}}_k, D_k, X_k)$ (Line 3) using Algorithm 4. We only consider the feasible links in the topology, e.g., the links with residual bandwidth $B_{e'}(u,v) \ge B_k, \forall e' \in E$.

If a solution exists, the corresponding path $\mathcal{P}_k$ is assigned for f_k (Line 6). However, if relaxing the bandwidth constraint does not return a path, we further relax the delay constraint by using Eq. (5.10), keeping the bandwidth constraint unmodified, and solve MCP_HEURISTIC$(N, s_k, t_k, \widetilde{\mathfrak{D}}_k, \mathfrak{B}_k, X_k, \widehat{B}_k)$ (Line 9). Once the path is found, we allocate the bandwidth for the scheduled flow and update the residual link bandwidth (Line 15). If the path is not found after *both* relaxation steps, the algorithm returns False (Line 17) since it is not possible to assign a path for f_k such that both delay and bandwidth constraints are satisfied. Note that the heuristic solution of the MCP depends of the parameter X_k. From our experiments we find that if a solution exists, the algorithm is able to find a path as long as $X_k \geq 10$.

5.5.2 Complexity Analysis

Note that Line 8 in Algorithm 4 is executed at most $(C_2 + 1)(V - 1)E$ times. Besides, if there exists a path, the worst-case complexity to extract the path is $|\mathcal{P}|C_2$. Therefore, time complexity of Algorithm 4 is $O(C_2(VE + |\mathcal{P}|)) = O(C_2VE)$. Hence the worst-case complexity (e.g., when both of the constraints need to be relaxed) to execute Algorithm 5 for each flow $f_k \in F$ is $O((X_k + \widehat{B}_k)VE)$.

5.6 Implementation

We implement our prototype as an *application that uses the northbound API* for the Ryu controller [59]. The prototype application accepts the specification of flows in the network. The flow specification contains the classification, bandwidth requirement and delay budget of each individual flow. In order for a given flow f_k to be realized in the network, the control-plane state of the network needs to be modified. The control-plane needs to route traffic along the path calculated for each f_k as described in Section 5.5. In this section, we describe how this is accomplished by decomposing the network-wide state modifications into a set of smaller control actions (called Intents) that occur at each switch.

5.6.1 Forwarding Intent Abstraction

An *intent* represents the *actions performed on a given packet at each individual switch*. Each flow f_k is decomposed into a set of intents as shown in Figure 5.3. The number of intents that are required to express actions that the network needs to perform (for packets in a flow) is the same as the number of switches on the flow path. Each intent is a tuple given by $(\mathsf{Match}, \mathsf{InputPort}, \mathsf{OutputPort}, \mathsf{Rate})$. Here, Match defines the set of packets that the intent applies to, $\mathsf{InputPort}$ and $\mathsf{OutputPort}$ are where the packet

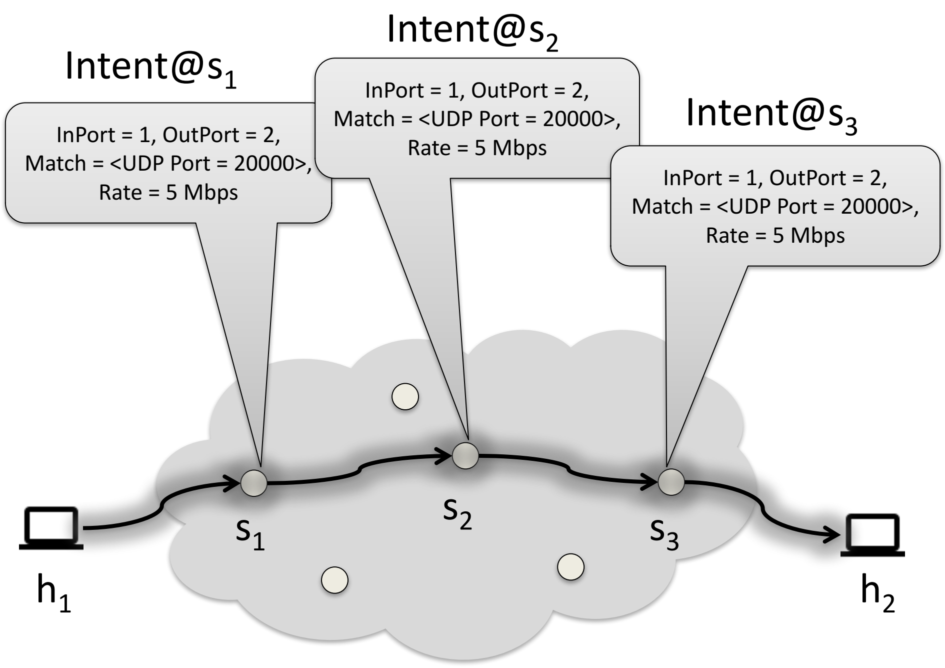


Figure 5.3: Illustration of decomposition of a flow f_k into a set of intents: f_k here is a flow from the source host h_1 to the host h_2 carrying mission-critical DNP3 packets with destination UDP port set to $20,000$. In this example, each switch that f_k traverses has exactly two ports.

arrives and leaves the switch, and finally the Rate is intended data rate for the packets matching the intent. In our implemented mechanism for laying down flow paths, each intent translates into a single OpenFlow [19] flow rule that is installed on the corresponding switch in the flow path.

5.6.2 Bandwidth Allocation for Intents

In order to guarantee bandwidth allocation for a given flow f_k, each one of its intents (at each switch) in the path must allocate the same amount of bandwidth. As described above, each intent maps to a flow rule and the flow rule can refer to a meter, queue or both. However, meters and queues are limited resources. Also, not all switch implementations provide both of them. As mentioned earlier (Section 5.3), we use the strategy of one queue per flow that guarantees better isolation among flows and results in stable delays.

5.6.3 Intent Realization

Each intent is realized by installing a corresponding flow rule by using the northbound API of the Ryu controller. Other than using the intent's Match and OutputPort, these flow rules refer to corresponding queues and/or meters. If meters are used, then they are also synthesized by using the controller API. However, OpenFlow does not support installation of queues in its controller-switch communication protocol, so the queues are installed separately by interfacing directly with the switches by using a switch API or command line interface.

5.7 Evaluation

The goal of the evaluation in this section is two-fold: *(a)* schedulability of a given set of flows across various topologies to explore the design space/performance of the path layout algorithm in Section 5.7.1, and *(b)* an empirical evaluation, using Mininet, that demonstrates the effectiveness of our end-to-end delay guaranteeing mechanisms even in the presence of other traffic in the network (Section 5.7.2).

5.7.1 Performance of the Path Layout Algorithms

Topology Setup and Parameters

In the first set of experiments we explore the design space (e.g., feasible delay requirements) with randomly generated network topologies and synthetic flows. For each of the experiments we randomly generate a graph with 5 switches and create $f_k \in [2, 20]$ flows. Each switch has 2 hosts connected to it. We assume that the bandwidth of each of the links $(u, v) \in E$ is 10 Mbps (e.g., IEEE 802.3t standard [88]). For this experiment the link delays are randomly generated within [5, 25] μs. For each randomly-generated topology, we consider the bandwidth requirement as $B_k \in [1, 5]$ Mbps, $\forall f_k$.

Results

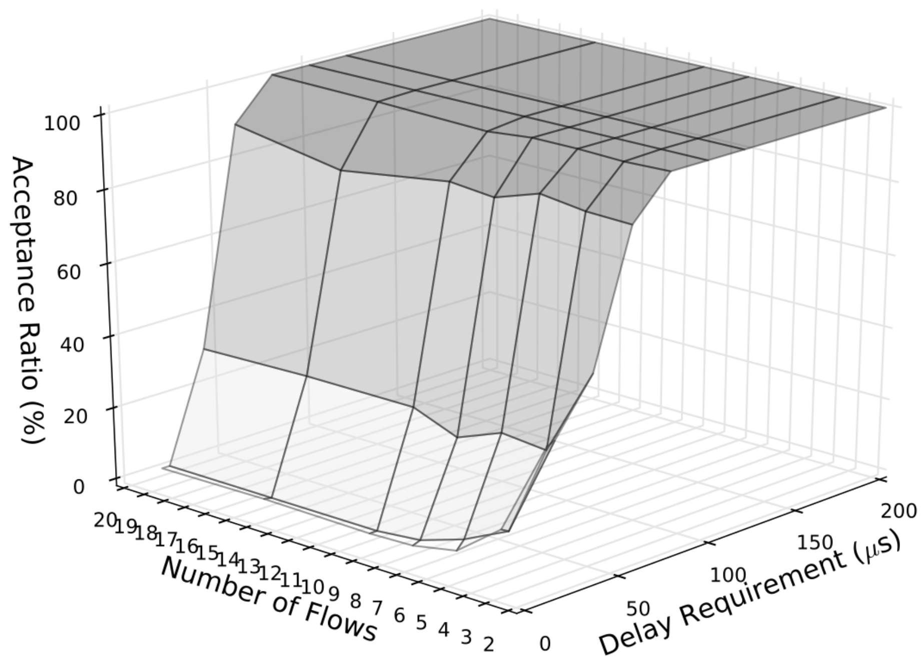


Figure 5.4: Schedulability of the flows in different network topology. For each pair (delay-requirement, number-of-flows), we randomly generate 250 different topologies. In other words, total 8 × 7 × 250 = 14,000 different topologies were tested in the experiments.

We say that a given network topology with set of flows is *schedulable* if all the real-time flows in the network can meet the delay and bandwidth requirements. We use the *acceptance ratio* metric (z-axis in Figure 5.4) to evaluate the schedulability of the flows. *The acceptance ratio is defined as the number of accepted topologies (e.g., the flows that satisfied bandwidth and delay constraints) over the total number of generated ones.* To observe the impact of delay budgets in different network topologies, we consider the end-to-end delay requirement D_k, $\forall f_k \in F$ as a function of the topology. In particular, for each randomly generated network topology G_i we set the minimum delay requirement for the highest priority flow as $D_{min} = \beta\delta_i$ μs, and increment it by $\frac{D_{min}}{10}$ for each of the remaining flows. Here δ_i is the diameter (e.g., maximum eccentricity of any vertex) of the graph G_i in the i-th spatial realization of the network topology, $\beta = \frac{D_{min}}{\delta_i}$ and D_{min} represents x-axis values of Figure 5.4. For each (delay-requirement, number-of-flows) pair, we randomly generate 250 different topologies and measure the acceptance ratios. As Figure 5.4 shows, stricter delay requirements (e.g., less than 60 μs for a set of 20 flows) limit the schedulability (e.g., only 60% of the topology is schedulable). Increasing the number of flows limits the available resources (e.g., bandwidth) and thus the algorithm is unable to find a path that satisfies the delay requirements of *all* the flows.

5.7.2 Emulation Experiments using Mininet

While the flow paths are laid out in a *correct-by-construction* manner (see Algorithm 5), our evaluation in this section tests our algorithms with a variety of cases to demonstrate that our delay-based admission control algorithms work as intended. This is akin to demonstrating the workings and checking the performance of a proven scheduling algorithm with synthetic task sets.

Experimental Setup

The purpose of the experiment is to evaluate whether our controller rules and queue configurations can provide isolation guarantees so that the real-time flows can meet their delay requirement in a practical setup.

We evaluate the performance of our proposed scheme using Mininet [57] (version 2.2.1), which has been widely used [89, 90, 91, 92, 93]. Mininet is an open source platform that *emulates* real-world setup by utilizing virtualization on top of a Linux kernel. Mininet has the capability to emulate

Table 5.1: Experimental platform and parameters

Artifact/Parameter	**Values**
Number of switches	5
Bandwidth of links	10 Mbps
Bandwidth requirement of a flow	[1, 5] Mbps
Controller	Ryu 4.7
Switch configuration	Open vSwitch 2.3.0
Network topology	Synthetic/Mininet 2.2.1
OS	Debian, kernel 3.13.0-100

different kinds of network elements such as host, switches (layer-2), routers (layer-3) and links.

We configured switches using Open vSwitch (OVS) [58] (version 2.3.0) and use Ryu [59] (version 4.7) as our controller. For each of the experiments we randomly generate a Mininet topology using the parameters described in Table 5.1.

We develop flow rules in the queues to enable connectivity among hosts in different switches. The packets belonging to the real-time flows are queued separately in individual queues and each queue is configured at a maximum rate of $B_k \in [1, 5]$ Mbps. If the host exceeds the configured maximum rate of B_k, our ingress policing throttles the traffic before it enters the switch.[1]

We use `netperf` (version 2.7.0)[78] to generate the UDP traffic between the source and destination for any flow f_k. Once the flow rules and queues are configured, we send packets from source s_k to host t_k for each of the flows f_k. The packets are sent in a burst of 5 with 1 ms interburst time. All packet flows are triggered simultaneously and last for 10 seconds.

To measure the effectiveness of our prototype with mixed (e.g., real-time and non-critical) flows, we enable [1,3] non-critical flows in the network. All of the low-criticality flows use a *separate, single queue* and are served in a FIFO manner – it is the "default" queue in OVS. Since many commercial switches (e.g., Pica8 P-3297, HPE FlexFabric 12900E, etc.) supports up to 8 queues per port (and 52 ports per switch), in our Mininet experiments we limit the maximum number of real-time flows to 7. We performed experiments for a single port where each of the 7 real-time flows uses a separate queue and the remaining 8th queue is used for non-critical flows. Our flow rules isolate the

[1]In real systems, the bandwidths allocation would be overprovisioned (as mentioned earlier), but our evaluation takes a conservative approach.

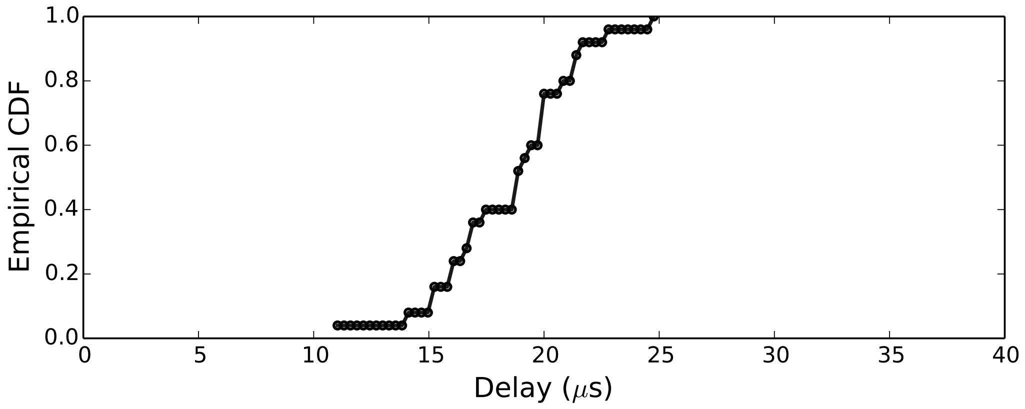
non-critical flows from real-time flows. All the experiments are performed on an Intel Xeon 2.40 GHz CPU and Linux kernel version 3.13.0-100.

We assume flows are indexed based on priority, i.e., $D_1 < D_2 < \cdots < D_{|F|}$ and randomly generate 25 different network topologies. We set $D_1 = 10\delta_i$ μs and increment with $\frac{D_1}{10}$ for each of the flow $f_k \in F, k > 1$ where δ_i is the diameter of the graph G_i in the i-th spatial realization of the network topology. For each topology, we randomly generate the traffic with required bandwidth $B_k \in [1, 5]$ Mbps and send packets between source (s_k) and destination (t_k) hosts for 5 times (each transmission lasts for 10 seconds) and log the worst-case round-trip delay experienced by any flow. We define the *expected delay bound* as the expected delay if the packets are routed through the diameter (i.e., the greatest distance between any pair of hosts) of the topology and given by $\mathfrak{D}_i(u, v) \times \delta_i$, where $\mathfrak{D}_i(u, v) = 5$ μs is the delay between the link (u, v) in i-th network realization. The link delay here is assumed to be a given property of the topology.

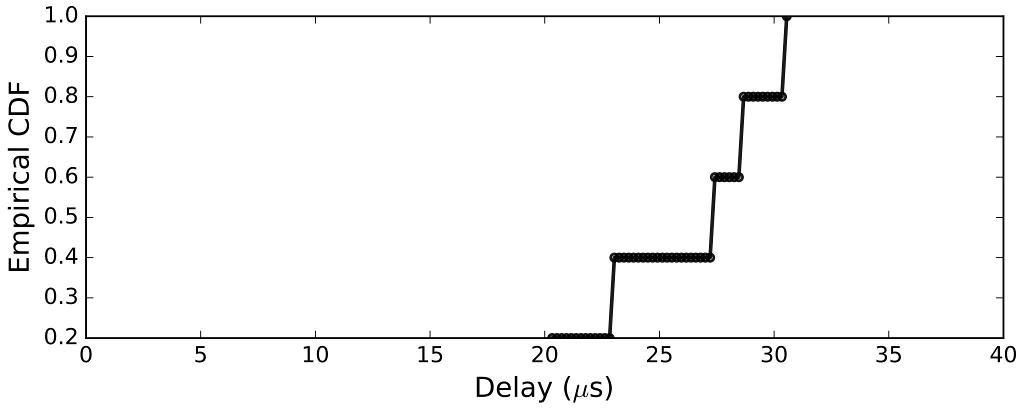
Experience and Evaluation

Recall that we use a correct-by-design principle to lay out the flows in the network. Figure 5.5(a) illustrates the results for the schedulable flows (viz., the set of flows for which *both* delay and bandwidth constraints are satisfied). The y-axis of Figure 5.5(a) represents the empirical CDF of average round-trip delay experienced by any flow. From our experiments we find that, the non-critical flows *do not* affect the delay experienced by the real-time flows and the average delay experienced by the real-time flows *always* meets their delay requirements. This is because our flow rules and queue configurations isolate the real-time flows from the non-critical traffic. As seen in Figure 5.5(a), the average round-trip delays are less than the maximum expected round-trip delay bound (*e.g.,* $2 \times 5 \times 4 = 40$ μs).

To compare, we conducted an experiment of laying out the same set of flows but *without* our mechanisms in place. This experiment used shortest-path routing and did not separate the queues for any flows (in contrast to the separate queues for real-time flows in our work). Figure 5.5(b) plots the empirical CDF of the mean delays experienced by the real-time flows in this setting. As the plot shows, these flows experienced higher and more variable latency than when our mechanisms were in place, thus highlighting the



(a)



(b)

Figure 5.5: The empirical CDF of: (a) average round-trip delay when using paths generated by MCP and giving each flow its own queue, worst-case round-trip delay; (b) average round-trip delay experienced when using shortest paths and a single queue. We set the number of flows $f_k = 7$ and examine $7 \times 25 \times 5$ packet flows (each for 10 seconds) to obtain the experimental traces.

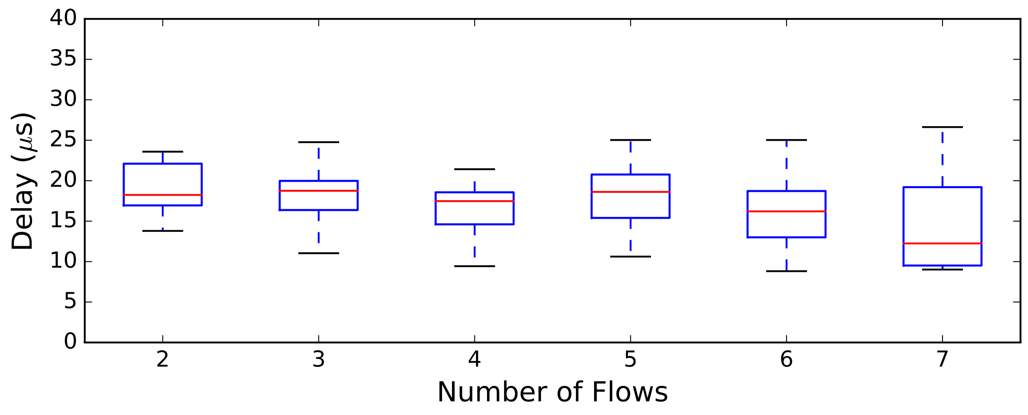


Figure 5.6: End-to-end average round-trip delay with varying number of flows. For each set of flow $f_k \in [2, 7]$, we examine $f_k \times 25 \times 5$ packet flows (each for 10 seconds). The blue boxes represent inter-quartile range (e.g., 50% of values for the group) while the inside red lines indicate median value. The upper and lower whiskers represent values outside the middle 50%.

need for the proposed mechanisms being presented in this paper. The 99th-percentile delays were also much higher than when using our mechanisms.

Figure 5.6 illustrates the impact of number of flows on the average round-trip delay (represented by y-axis in the figure) with different number of flows (x-axis). Recall that in our experimental setup we assume at most 8 queues per port are available in the switches where 7 real-time flows are assigned to each of 7 queues and the other queue is used for [1, 3] non-critical flows. As shown in Figure 5.6, increasing the number of flows slightly decreases quality of experience (in terms of end-to-end delays). With increasing number of packet flows the switches are simultaneously processing forwarding rules received from the controller – hence increasing the round-trip delay. Recall that the packets of a flow are sent in a bursty manner using `netperf`. Increasing number of flows in the Mininet topology increases the packet loss and thus causes higher delay.

CHAPTER 6

SYNTHESIS: RESILIENCY GUARANTEES

The applications that constitute the critical infrastructure (e.g. smart power generation and distribution systems, oil refineries etc.) have a unique set of requirements regarding their underlying communication networks. For example, such applications require that their communication be *seamlessly* resilient against *arbitrary* link or device failures. Furthermore, these applications also require a *predictable* end-to-end delay for data delivery in multicast settings [10] [94]. Such resilience and performance requirements cannot be simultaneously accomplished by mere overprovisioning of network resources such as topological redundancy or bandwidth.

Rather, in the packet store-and-forward paradigm, the resiliency can be provided by carefully routing the packets around a failed link or network device and requires solving complex combinatorial problems [95] [96] [97] [98]. However, even with the use of fast-failover mechanism, it is impossible to provide seamless resiliency using such techniques due to a finite delay caused by a switch's capability to detect and respond to failure and change in routing.

Such intractability is a result of *hard* routing and resource allocation decisions that are in turn a consequence of the atomic nature of a packet flow in the store-and-forward paradigm. In this paradigm, a flow has to originate at a source port and follow a specific path to arrive at the destination(s) without any modifications to its contents. However, network coding converts this hard decision into one of many *soft* decisions by mixing packets at intermediate network devices using algebraic coding. In theory, NC promises to provide seamless resilience to failures for critical infrastructure applications over the store-and-forward paradigm [27] [28]. However, practical NC that achieves the promised theoretical gains has remained elusive.

Clearly, NC is realized when the intermediate network devices can be programmed to implement the packet coding and decoding capabilities. While there have been successful attempts to demonstrate the efficacy of using inter-session NC in wireless networks [99] [100], the progress on the widespread adoption of the same has been disappointing. In part, the reason has been the practical issues of retrofitting NC onto the prevalent networking architecture. These issues have been addressed in various ingenious efforts in the past [101] [102]. But, more importantly, the adoption of NC has been stifled due to a lack of programmable platforms that can implement novel data-plane methods at scale. Historically, the switch ASIC architectures that implement data-plane functionality have been optimized for ever-increasing line-speed performance at the expense of programmability. However, very recently, with the advent of programmable data-planes [20], it has become possible to not only experiment with [103] but also to deploy new network functions using a flexible data-plane architecture in production networks [104].

Based on these developments, we devise an architecture capable of simultaneously meeting resilience and performance requirements of the data streams generated by applications in critical infrastructure systems. To that end, we present one that leverages programmable networks to replace routing algorithms with NC functions. Our contributions include:

- A library of atomic network coding primitives implemented using the programmable data-planes.
- Use of the proposed primitives to construct linear network coding functions capable of achieving specific requirements for applications' data streams.
- Evaluation of the coding functions to show that the seamless resilience and multicast rate gains are obtained at a small per-packet processing cost of coding and decoding the packets in the data-plane.

The remainder of this chapter is organized as follows: Section 6.1 discusses some experiments that motivate the development of network coding for providing seamless resiliency; Section 6.2 discusses related work; Section 6.3 provides some background; Section 6.4 proposes an architecture to implement coding functions; Section 6.5 discusses the design of various elements of the proposed architecture; Section 6.6 evaluates the performance and costs of using the proposed design.

6.1 Motivating Experiments

As noted above, applications in the CI networks need forwarding resilience to failure events that cause the network topology to change due to link or device failures. This resilience implies that the packets in a given set of flows continue to be delivered while the event occurs. When a distributed control-plane is used, such an event results in routing table updates and consequently the packets that are in-transit are lost. Similarly, even with the centralized control-plane and the use of fast-failover mechanism, there are packet losses due to the time taken by the switch to shift traffic from one link to another.

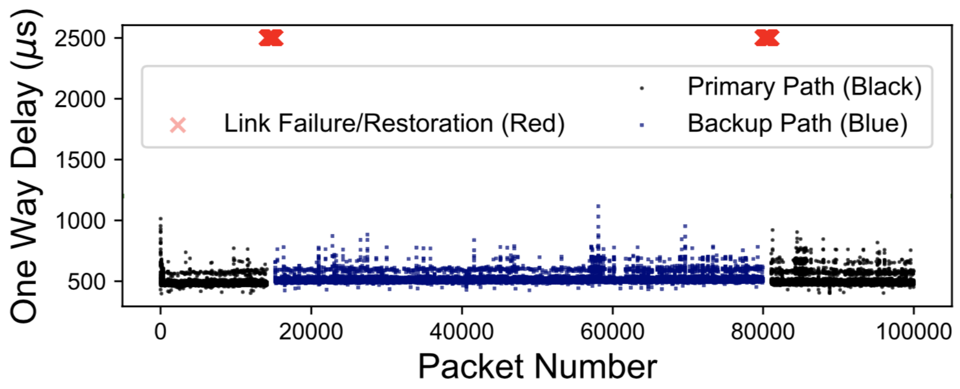


Figure 6.1: Packet drops due to the use of fast-failover mechanism: 100,000 packets were sent at the rate of 95 Mbps.

Table 6.1: Number of packets lost when fast-failover mechanism is engaged due to a link failure on an OpenFlow compatible hardware switch

Rate (Mbps)	**# Packets Lost**
4	15
10	71
20	115
50	321

We performed an experiment to motivate using Network Coding to provide seamless resilience. We used an OpenFlow enabled switch (Pica8 P-3297) and three hosts (Raspberry Pi 3 Model B). We set up one of the hosts as the packet source and other two hosts as receivers. We configured the switch to forward the packets from the source to one of the receivers. However, we use the fast-failover mechanism such that when link to one of the receivers fails, the packets are forwarded to the other receiver. We sent 100,000 packets with 1 kB payload with fixed inter-packet period to generate different data rates and manually failed one of the receiver links. We measured the total number of packets received at both hosts. Table 6.1 presents the total number of packets lost at varying rates. We observed the number of lost packets to increase with the data rate of the sender.

Furthermore, we performed a similar experiment with a higher number of switches and only two hosts: a source and a destination. We sent 100,000 packets from the source to the destination and measured the end-to-end delay of each packet. We failed and restored the first link in the primary path of the packet flow. The primary path of the packets contained three switches. When a link failure occurred, the path became longer with four switches. Figure 6.1 plots the end-to-end delay for each packet. We observed that approximately 1000 packets were dropped (shown in red) when both the link failure and restoration events occurred.

It is clear that more packets are lost at higher flow send rates. The results from both Table 6.1 and the Figure 6.1 correspond with a link failure/restoration time of about 100 μs. This is consistent with what has been previously reported for some specially designed switches as well [105]. The takeaway from these experiments is that it is hard to escape packet losses even with fast-failover enabled OpenFlow switches.

6.2 Related Work

There has been prior work in the store and forward paradigm that allows nearly instantaneous failure recovery. When such failures are addressed reactively, they result in prohibitively large restoration time for critical infrastructure applications [106] [10]. There are proactive approaches to deal with such failures which use the mechanisms local to a switch to reroute traffic on an alternative path [95] [96] [97] [98]. However, such approaches require k-connected network topologies for sustaining k link failures, thus incurring a large overhead in procuring and maintaining such networks. Furthermore, these approaches require solving combinatorial problems to choose alternative links in the event of link failures. These approaches also lead to new problems such as the need to *load-balance* resilience so that a small set of links does not become too critical for the resulting network after the failures.

Recently, in order to meet the performance guarantees in store-and-forward networks, the standards bodies have proposed standards for special-purpose hardware [12] [13]. However, using such special hardware incurs large capital and recurring expenses. To that end, some recent work has proposed mechanisms to simultaneously meet per-flow end-to-end delay and bandwidth requirements using software-defined commodity networks [107]. However, this work solves a resource allocation problem using a heuristic for a multi-constraint path problem that provides no guarantees of optimality.

The seminal work that demonstrated a practical mechanism to implement NC by using simulations was done by Chou et al. [102]. This work focused on coding batches of data which is incompatible with the acknowledgement mechanisms of TCP. Subsequently, there has been work demonstrating TCP throughput gains with the use of NC [101] [108] with deployable implementations. However, these efforts focus on intra-session coding at source only and the goodput gains obtained due to intermittent packet loss.

There have been several prior efforts to implement NC on top of the application layer, either in an overlay topology [109] [110] [111] or as a virtual network function [112]. While implementing NC in the application layer offers flexibility and variety in the type applications that can be materialized, the cost of taking packets from the network interface and processing them in upper layers can be high and can be mitigated by implementing NC in the data-plane of network devices. Furthermore, COPE [99] demonstrated the benefits of inter-session coding in the specific setting of wireless networks by utilizing the broadcast property of the media with a clever heuristic. The types of benefits that COPE extracted using a specially designed architecture can now be replicated for wired networks by implementing NC using standard platforms such as the P4 ecosystem.

There have previous theoretical proposals of using coding in the CI systems [113] [114] [115]. In practice, such proposals have focused on increasing throughput for collecting wireless sensor measurements [100].

Finally, there are priors efforts to code packets at a link level to improve robustness. Specifically, when optical links suffer grey failures, the links throughput can be improved by using forward error correction (FEC) codes [116] or simple parity codes [117]. However, there are no multicast throughput gains obtained by using coding across an individual link.

6.3 Network Coding

In the store-and-forward paradigm, each flow originates at a source port and follows a deterministic path to arrive at the destination port(s); however, its contents are immutable during the transit. Hence, due to immutability, when globally optimal decisions for resource allocation for flows are to be made, the resources are allocated separately for each flow at the network devices. Furthermore, in the event of a link or device failure, the flows must be routed around the failure entirely. Due to these requirements of immutability, solving for performance guarantees and resilience requirements results in formulation of problems that are intractable [118] [107] or combinatorially complex [95] [96] [97] [98].

The NC paradigm approaches the problem of delivering data from point A to point B by allowing intermediate nodes within the network to code and recode the packets. This paradigm has many promising theoretical properties. For example, in their seminal work, Ahlswede et al. [27] showed that, given a network represented as a multigraph $G(V, E)$, network coding can enable a sender $s \in V$ to communicate with a set of receivers $T \subset V \setminus s$, at a multicast rate equal to the minimum max-flow from the sender to any of the $t \in T$. Li et al. [119] showed that using linear codes on the network nodes is sufficient to achieve this rate. Koetter and Medard [28] extended the theorem to the cases when the edges in E are subject to failures and showed that the linear codes can achieve minimum max-flow even after failures. Finally, Ho et al. [120] showed that the random linear network codes suffice to achieve the same.

The implementation of linear NC requires two types of computations: First, there are network-level operations such as computation of coding coefficients or designation of various roles to the individual nodes based on the topology and the application requirements. These operations can be performed in the programmable control-plane. Second, there are the simple arithmetic operations (e.g. addition, multiplication) that are performed on an individual or a small batch of packets in the applications' data stream. In order for coding to scale to line-speeds, these operations have to be performed on the individual network devices using a programmable data-plane architecture.

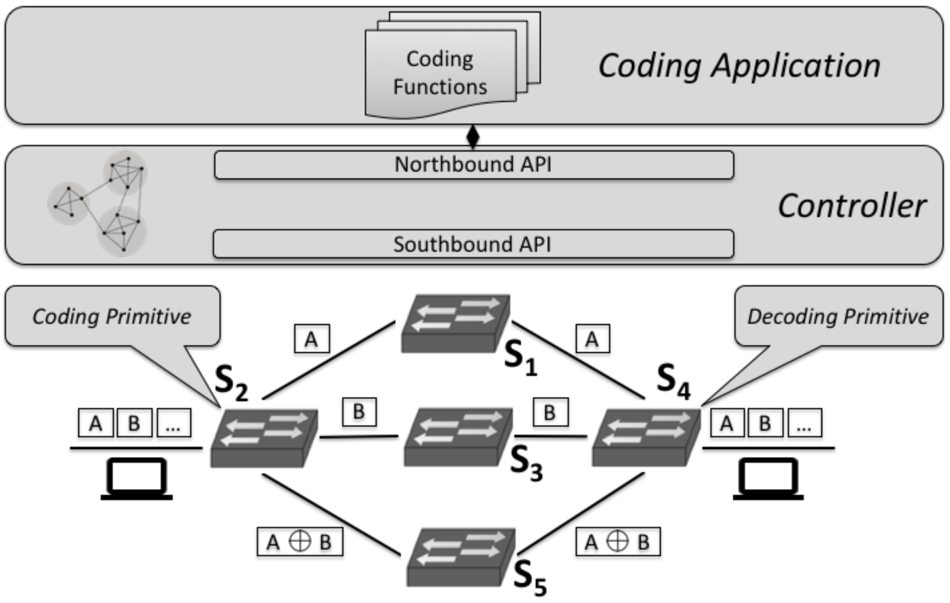


Figure 6.2: Architecture.

6.4 Architecture

Figure 6.2 illustrates the proposed architecture which enables implementation of linear network coding using P4 devices. This particular example shows a stream of packets carrying applications' data originating at the host on the left side and terminating on the host at the right side. Fundamentally, we assume that a packet stream can be divided into a batches of packets. These batches are then processed by individual devices to achieve the NC gains.

We define a *coding function* as the realization of a linear code to improve resilience or throughput of a unicast/multicast data stream. For example, in Figure 6.2, the function implements a diversity code [121] to provide resilience to failure of any one of the three paths between S_1 and S_3. The function replaces IP forwarding and spans one or more P4 enabled devices. A northbound coding application implements multiple coding functions that operate simultaneously across the network.

A *coding primitive* is an atomic block of functionality implemented on the individual P4 switches. For example, in Figure 6.2, switches S_1 and S_3 implement the coding and decoding primitives respectively. Each primitive operates independently of the others. Each incident stream of packets on the device is subject to one or more primitives. A switch can process multiple data streams simultaneously. Each switch's configuration contains the

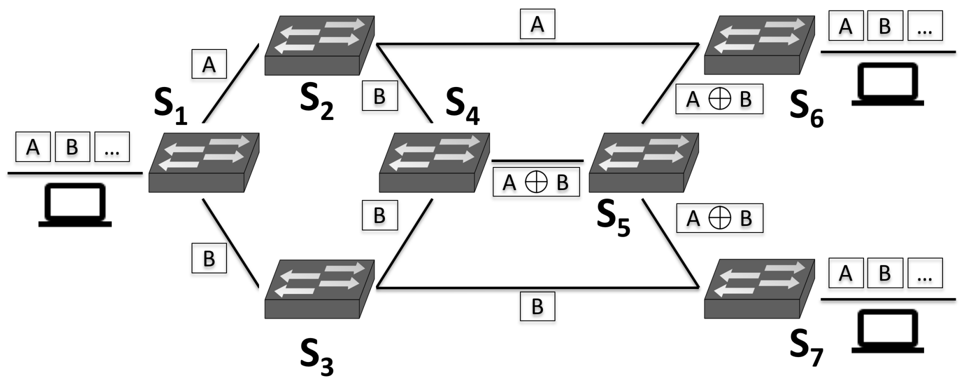


Figure 6.3: Coding function for realizing multicast rate gains over a *butterfly* topology: The host on the left (at switch S_1) is sending a multicast stream to the two hosts on the right at switches S_6 and S_7.

identifier for the data streams and the exact sequence of coding primitives applied to each of them.

6.5 Design

In this section, we present our design for coding functions over the Galois Field of size 2 (GF2). While the benefits of using coding over this field are less than larger complex field sizes, our proposed design applies to larger field sizes and captures both the benefits and requirements of feasible coding implementations. We first describe our design for the coding functions in the control-plane. Next, we discuss the packet header that is used to coordinate primitives for a given coding function and finally describe how our design of the coding primitives in the data-plane uses the P4 ecosystem.

6.5.1 Coding Functions

The coding functions are a part of the coding application. Each coding function takes as input the source host and destination host(s) associated with the data stream. It accesses the topology information by using the controller's northbound API. Then, the coding function generates the configuration for the coding primitives described later in this section. Figures 6.3 and 6.4 show instances of coding functions. One instance is that of a diversity code that provides seamless resilience for a unicast stream over three paths. The other

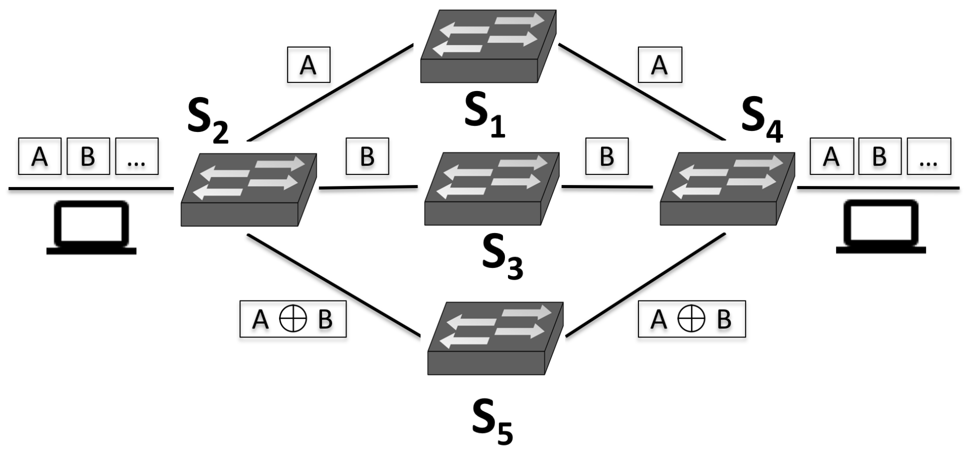


Figure 6.4: Coding function for realizing seamless resilience to failure of links in any one of three available paths from host on the left (at switch S_2) to the host on the right (at switch S_4).

instance uses a linear code for enhancing receiver's data rate of a multicast stream.

6.5.2 Coding Header

Each packet that belongs to a coding function carries a coding header. The header contains various fields to coordinate the operations performed by the coding primitives across the network. The header has a field called `next_primitive` which determines what happens to the packet when it arrives at a network device. It also has a field called `stream_id` to identify packets belonging to different streams. Finally, it has the `batch_number` which identifies the packets belonging to a given batch of packets within the stream.

Since P4 does not provide access to the contents of a packet, in our prototype, we use a field in the coding header to carry the packet's payload. However, the number of bytes in each packet that can be manipulated as the packet header are a fraction of the total packet size. The exact number of bytes varies based on the design of the P4 switch but it is not beyond a few hundred bytes. While, in practice, CI application protocols have the ability to adjust the size of their payloads by trading them with sampling frequency, this remains a fundamental limitation of the commercially available hardware switches. Furthermore, it is worth mentioning that there are no fundamental limitations on the software switches and they can choose to explore the entirety of the packet payload as a header albeit with an increase in the per-packet processing cost.

The coding primitives are implemented in the data-plane on the individual network devices using P4. The primitives are implemented primarily in the ingress pipeline. However, some of the primitives use the egress pipeline for recirculating cloned packets for generating new packets that carry coded/decoded payload. For every primitive, we also collect some in-band telemetry to measure processing times for evaluation.

Coding primitives use several common design patterns. Each primitive uses at least one table in the ingress pipeline. If the primitive uses packet cloning and recirculation, then it also uses a table in the egress pipeline. Furthermore, each primitive table has a common field called `stream_id` as part of its `key`. This field is used to specify the packets belonging to a specific application's data stream. These packets could originate at the host or could be the result of the output of another primitive.

Each table implements a decision tree. The levels of the tree are determined by values taken by the fields in the `key` of the table. The actions in the table form the leaves in the decision tree. These actions either perform the mathematical operation for coding/decoding packets or manipulate some global state that is held in registers.

6.5.3 Coding Primitives

Next, we describe the particulars of the individual primitives. As illustrated in the Figure 6.5, we developed five primitives to implement linear inter-session NC as follows:

- **Splitting:** Primitive splits a given packet stream arriving from a single interface into individual batches of packets. It uses global state in registers on a per-stream basis and stores the packets in the appropriate registers so that the coding/decoding primitives can use them.

- **Coding:** Primitive *generates* new packets whose payloads are obtained by coding over the previously stored payloads. It accomplishes that by using the cloning and recirculation features to create a loop to generate packets whose payload is then populated to be the coded packets.

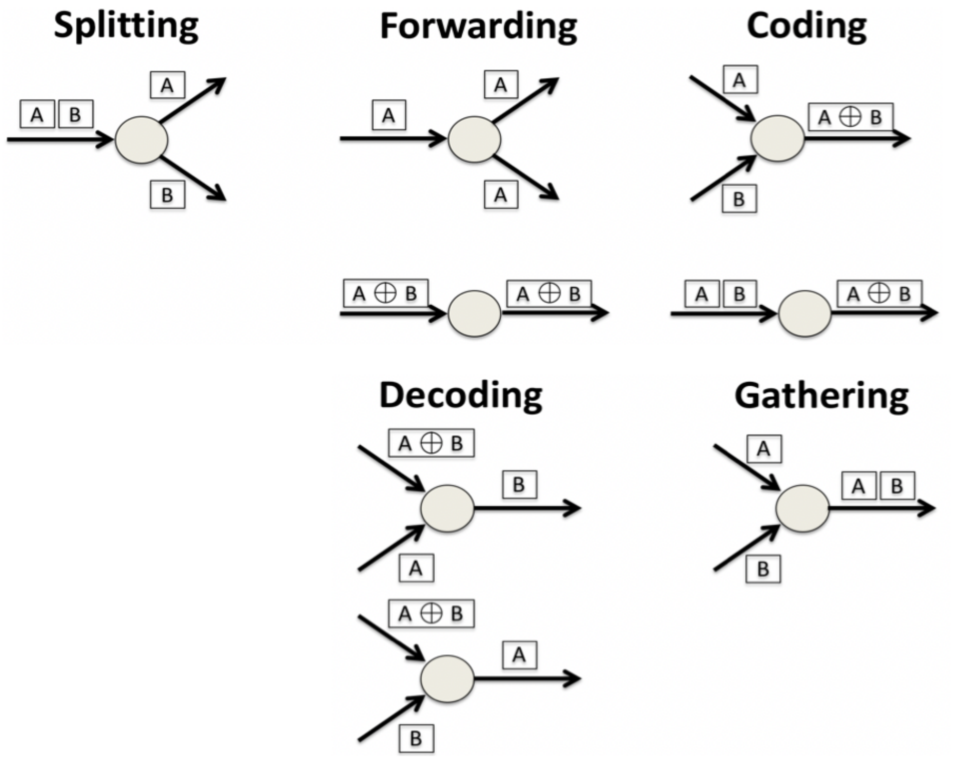


Figure 6.5: Network coding primitives.

In order implement linear codes, the coding primitive only performs addition, subtraction and XOR operations provided by P4.

- **Forwarding:** Primitive performs unicast or multicast forwarding of a packet. The multicast forwarding action makes use of cloning to generate copies of packets.
- **Gathering:** Primitive collects a batch of incoming packets from multiple interfaces and puts them into the registers corresponding to their `stream_id`. It also relies on global state in registers to keep track of the packets it has received on a per-stream basis.
- **Decoding:** Primitive takes the gathered packets, decodes them and forwards the payload packets to the host. In order to generate decoded payload, this operation may also require *generating* new packets. This primitive also uses cloning and recirculation to generate new packets that are then populated with decoded payload.

Clearly, the coding and decoding primitives require buffering of the packets. In our current prototype, we use the registers to maintain a fixed size ring buffer containing the packet payloads and use the `batch_number` to locate them appropriately. The packets belonging to same stream of packets are correlated by using the `stream_id`. In our design, we do not perform any flow control on the switches.

Furthermore, one of the key concerns associated with the design of both coding and decoding mechanism design is the need to wait for the arrival of packets in the buffer at a primitive before they are ready to code or decode a collection of packets. In our design, we avoid waiting at coding and send packets out as soon as they arrive. However, at decoding, especially when the first received packet is a coded packet, then the wait is inevitable. In such cases, the difference in the delay of a coded packet vs. a plain packet becomes a bottleneck. We evaluate various scenarios to explore this further in Section 6.6.

6.6 Evaluation

We implemented a prototype of the library of primitives and functions that use them. Our prototype is available in the public domain [122]. We evaluated our approach using `mininet` [57] and a software switch [123] as P4 target. The end-hosts were emulated using Python scripts that used `scapy` [124] to construct and parse custom coding headers. The emulations were performed on a machine that was running Ubuntu 16.04 LTS. The machine had eight processor cores clocked at 2.7 GHz and 16 GB of RAM. We performed two types of evaluation that are described below.

6.6.1 Multicast Rate Gains of Coding

We performed an experiment to measure the multicast rate gains that are obtained when using a simple linear code to perform multicast over the classic butterfly network shown in Figure 6.3. The host on the left side wants to multicast a data stream to the two hosts on the right. Suppose the bandwidth of the links between the switches is k bps. Theoretically, coding should allow a multicast rate of k bps for flows $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$ simultaneously, whereas any scheme that uses packet forwarding would not be able to

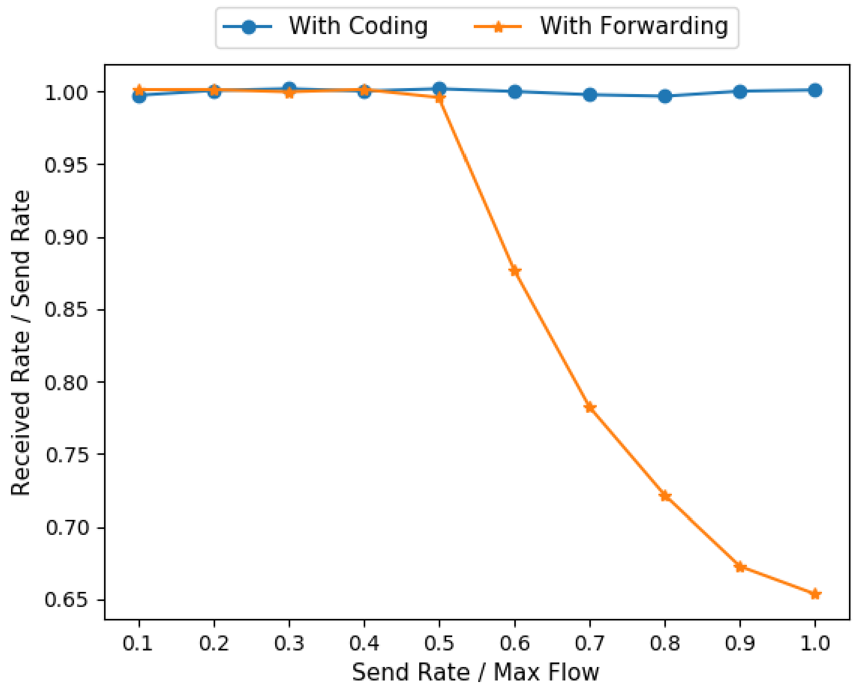


Figure 6.6: Comparison between received rate achieved using coding vs. forwarding in the butterfly topology.

accomplish this rate because only a single packet can be forwarded along the link $S_4 - S_5$.

We sent 750 packets at increasing data send rates with an exponentially distributed inter-packet time to match the rate. Each packet contained 4096 bytes of payload. For the purpose of this experiment, the value of k was set to 0.05 Mbps. We measured the simultaneous received data rate on each receiver by using the packet payload sizes and their timestamps in the generated PCAP files to obtain the range of time for which the transmission was received at each receiver. Figure 6.6 plots two ratios. The x-axis is the ratio of send-rate to the max-flow between the source and destination host, whereas the y-axis is the ratio of observed received rate at one of the receiving hosts (without loss of generality and empirically identical) to the send rate.

We observe that the received rate for forwarding starts to drop when the send rate is at 50% of the max-flow. This is because forwarding cannot entirely use the max flow bandwidth. However, the received rate for the case when coding is used does not drop at all. Furthermore, notice the difference

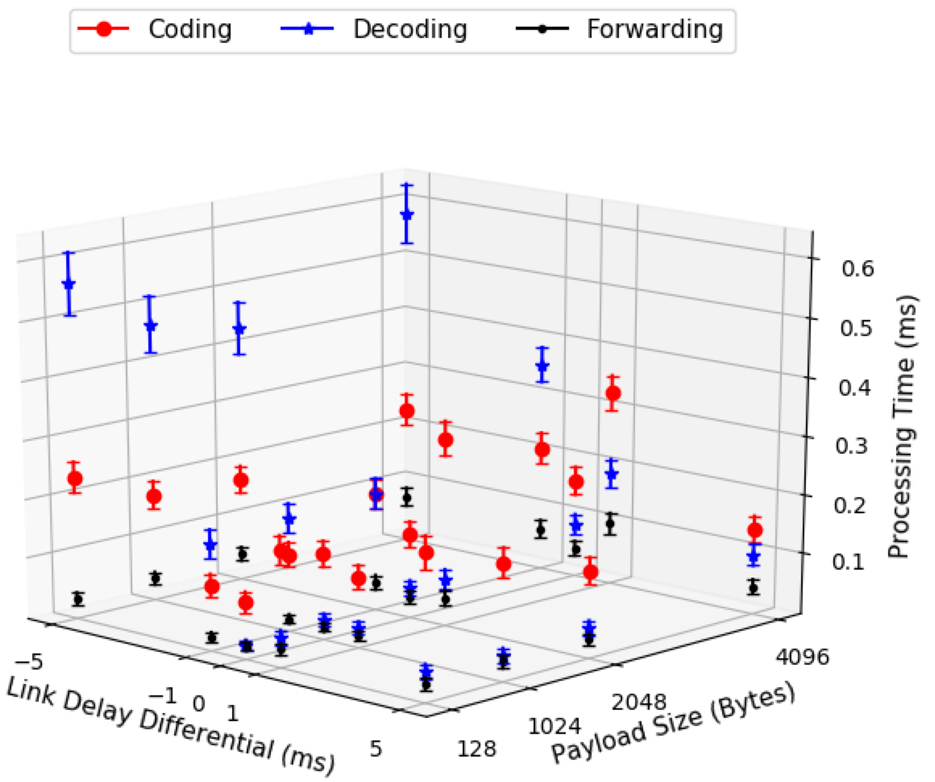


Figure 6.7: Processing time per packet for coding and decoding using the diversity code.

between received rate for both cases when the data send rate is the same as max flow. This relative gain in the received rate for coding is consistent with the a 33% gain when using coding over forwarding for a butterfly topology as theorized by Ahlswede et al. in their seminal paper.

6.6.2 Microbenchmarks

We performed an experiment to measure the processing latency associated with coding and decoding packets in a P4-enabled switch. We used in-band telemetry to measure the processing time at each switch in the path of the packet. The processing time is measured as difference between the time-stamp associated with packet arriving at the ingress pipeline and packet being queued for egress.

We used the framework described above to implement diversity coding over multiple alternative paths as shown in Figure 6.4. Each link was set to have a delay of 5 ms. However, in order to create a delay differential for packets arriving at S_5 for decoding, we changed the delay for link between S_2 and S_5. Furthermore, we measured the impact of payload size on the processing time. We sent 1000 packets for each point in the plot shown in Figure 6.7. The packets were sent as fast as possible (i.e. there was no sleep between any two packets).

We observed a negligible effect of increasing the payload sizes of packets for all operations. Specifically, we observed that the processing time for the set of switches that forward the packets (i.e. S_1, S_3, S_5) increases only slightly even when payload sizes are quadrupled. In the worst case, we observe that processing time at the coding node (i.e. S_2) can be up to four times the processing time for only forwarding the packets. Similarly, in the worst-case, a decoding node (i.e. S_4) can take up to six times longer to process a packet than a forwarding node. Some of this variation and extra processing time is due to the cloning and recirculation operation that coding and decoding primitives use.

Finally, we observe that for a lower link delay differential, the decoding time is higher than the coding time and has a high standard deviation, and vice versa. This is because different sets of table actions are in effect in those two cases. For a lower link delay differential, the XOR packet arrives at the decoder first and necessitates the use of arithmetic decoding, whereas when the differential is higher, the decoding is essentially reduced to forwarding the uncoded packets.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Summary of Dissertation Research

Network programmability has profound implications for the design and operations of the networks in CI systems. These networks have unique requirements of guarantees around achieving end-to-end delay, access control and resiliency. This dissertation is an attempt to design and prototype a set of tools that exploit network programmability to provide such guarantees. We develop these tools by proposing new architectures, algorithms and models. We developed prototypes for these tools and evaluated them for their usability and their efficacy towards providing the pertinent guarantee. We took two distinct approaches in development of these tools: analytic and synthetic.

In traditional, distributed data-plane networks, computationally analyzing the network configuration was challenged by the logistics: The configuration language for network devices was distributed, opaque and non-standard. However, with the rise of standardized control and data-planes, the computation analysis of network configurations has emerged as a powerful approach for solving a variety of problems. We used this approach to validate access control and resilience of networks simultaneously. We also used it to provide a mechanism to compute metrics pertaining to network resilience by using Monte Carlo methods.

In traditional networks, synthesizing network configuration so that it met a certain network policy goal was achieved indirectly by using a distributed routing algorithm. However, the mechanisms that were used were error-prone and neither provided the room to innovate nor optimally used the network resources. However, with the use of programmability, we synthesized control-plane configuration in the CI networks to meet their resilience, and end-to-end delay requirements have become possible. We also used centralized network coding based mechanisms that operate on an intersection of control and data planes to provide seamless resilience.

7.2 Future Work

The long-term goal of our research is to use programmable networking in the CI systems. We briefly described several ways in which this dissertation's work can be extended to help support this goal.

7.2.1 Analysis

To perform policy validation using the RRP, we proposed the use of header spaces combined with a port graph model of the underlying network. However, the underlying assumption was that the network is composed of OpenFlow devices. The port graph model itself is applicable towards programmable networks composed of P4-enabled devices. However, in order to validate policies on such programmable networks using FlowValidator, both the data-plane programs as well as the current configuration of the device need to be taken into account. So, a future research extension could explore whether such models can accurately capture the arbitrary behaviors that a programmable switch may implement. Such future work could also explore whether there is fundamental trade-off between model expressiveness and the associated computational complexity for use of the same to perform policy validation query expressed in RRP.

To perform resiliency metric computation, we proposed explicitly simulating the data-plane to enable Monte Carlo methods. This results in constraints on the nature of questions that can be asked about the network; however, it does offer a lot better performance with the use of multi-threaded implementations. While our work focused on resiliency, the programmable data-planes are being proposed to solve a variety of problems (e.g. to detect DDoS attacks [125] or to perform in-network consensus [126]). It is not a priori clear whether these applications are feasible and the behavior of the programmable data-planes may be simulated to answer questions around performance of such potential future applications. While the proposed data-plane simulator is specifically designed to mimic a network composed of OpenFlow switches, there is no fundamental limitation on its extension towards simulating P4-enabled data-planes.

7.2.2 Synthesis

To provide delay and bandwidth guarantees, we proposed algorithms to search, and a mechanism to synthesize, configurations for paths. However, our proposal makes a simplifying assumption to use a dedicated queue for each flow at each port in the network. This may not be scalable because the number of queues is fixed in hardware switches. This calls for future work that multiplexes flows onto a single hardware. However, this would necessitate the use of more sophisticated models (such as the ones rooted in Network Calculus [79]) to accurately capture the scheduling of packets in the switch processing pipeline and ports. Furthermore, it needs to be explored whether a similar approach can be used to provide both fast failover and end-to-end delay guarantees simultaneously. The naive approach to solving the joint problem will require solving many instances of the same problem by removing the presumably failed link from the topology. However, the naive approach would not scale well for large topologies.

To provide resiliency guarantees, we proposed a framework that uses of network coding. Our current implementation uses simple codes that provide fairly robust resilience for various scenarios. However, we have barely scratched the surface of the opportunities that more sophisticated codes can provide, and more sophisticated coding requires better inherent support from the underlying switch hardware. Our current implementation relies on using the P4 language's `clone` and `recirculate` primitives to generate new packets to mimic a loop. However, the hardware switches are not designed for such use. There is also no explicit looping mechanisms in the P4 language, forcing us to rely on the P4 enabled software switch implementation in our evaluation. The hardware based implementation on a programmable switch remains a work in progress.

Furthermore, beyond the raw implementation issues, as evident in our evaluation, there are processing costs to coding in the data-plane. This may not be suitable for delay-sensitive CI applications. It might be worthwhile to decide to not code packets if it is not necessary to provide resiliency. Hence, there is an opportunity for future work to explore the trade-off between coding choices with per-packet scheduling decisions. Some recent work has demonstrated the use of reinforcement learning (RL) to schedule packets in a broadcast cellular network [127] [128]. RL relies on using feedback from the system to adjust some parameters on its policy. With the rise of in-band telemetry (INT) [129], getting per-packet delay is now possible. This delay could be used to guide an RL agent that makes appropriate trade-offs based on the agent's objective in the context of CI networks.

REFERENCES

[1] J. Moteff, C. Copeland, and J. Fischer, "Critical infrastructures: What makes an infrastructure critical?" Library of Congress Congressional Research Service, Washington DC, 2003.

[2] Department of Homeland Security, "Critical Infrastructure Sectors," https://www.dhs.gov/critical-infrastructure-sectors, 2019, [Online; accessed November, 2019].

[3] Public Safety Canada, "Critical Infrastructure," https://www.publicsafety.gc.ca/cnt/ntnl-scrt/crtcl-nfrstrctr/index-en.aspx, 2019, [Online; accessed November, 2019].

[4] G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou, I. Kamwa, P. Kundur, N. Martins, J. Paserba, P. Pourbeik, J. Sanchez-Gasca et al., "Causes of the 2003 major grid blackouts in North America and Europe, and recommended means to improve system dynamic performance," *IEEE Transactions on Power Systems*, vol. 20, no. 4, pp. 1922–1928, 2005.

[5] Electricity Information Sharing and Analysis Center (E-ISAC), "Analysis of the cyber attack on the Ukrainian power grid," 2016.

[6] S. Karnouskos, "Stuxnet worm impact on industrial cyber-physical system security," in *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society*. IEEE, 2011, pp. 4490–4494.

[7] North American Electric Reliability Corporation, "Cyber Security – Electronic Security Perimeter(s)," https://www.nerc.com/pa/Stand/pages/cipstandards.aspx, 2019, [Online; accessed November, 2019].

[8] Department of Health and Human Services, "The HIPAA Security Rule," https://www.hhs.gov/hipaa/for-professionals/security/index.html, 2019, [Online; accessed November, 2019].

[9] A. Wool, "Trends in firewall configuration errors: Measuring the holes in Swiss cheese," *IEEE Internet Computing*, vol. 14, no. 4, pp. 58–65, 2010.

[10] D. E. Bakken, A. Bose, C. H. Hauser, D. E. Whitehead, and G. C. Zweigle, "Smart generation and transmission with coherent, real-time data," *Proceedings of the IEEE*, vol. 99, no. 6, pp. 928–951, 2011.

[11] C. M. Fuchs, "The evolution of avionics networks from ARINC 429 to AFDX," *Innovative Internet Technologies and Mobile Communications (IITM), and Aerospace Networks (AN)*, vol. 65, 2012.

[12] IEEE C/LM - LAN/MAN Standards Committee, "IEEE standard for local and metropolitan area networks – bridges and bridged networks – amendment 26: Frame preemption," *IEEE Std 802.1Qbu-2016 (Amendment to IEEE Std 802.1Q-2014)*, pp. 1–52, Aug 2016.

[13] M. D. J. Teener, A. N. Fredette, C. Boiger, P. Klein, C. Gunther, D. Olsen, and K. Stanton, "Heterogeneous networks for audio and video: Using IEEE 802.1 audio video bridging," *Proceedings of the IEEE*, vol. 101, no. 11, pp. 2339–2354, 2013.

[14] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the internet," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 1, pp. 59–64, 2003.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[16] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking control of the enterprise," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 1–12.

[17] The ONOS project, "ONOS," https://www.onosproject.org/, 2019, [Online; accessed November, 2019].

[18] The OpenDaylight Foundation, "OpenDaylight," https://www.opendaylight.org/, 2019, [Online; accessed November, 2019].

[19] Open Networking Foundation, "OpenFlow Switch Specification 1.3," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, 2019, [Online; accessed November, 2019].

[20] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese et al., "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[21] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano et al., "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 183–197, 2015.

[22] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu et al., "B4: Experience with a globally-deployed software defined WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 3–14.

[23] A. Gupta, R. MacDavid, R. Birkner, M. Canini, N. Feamster, J. Rexford, and L. Vanbever, "An industrial-scale software defined internet exchange point," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 1–14.

[24] North American Electric Reliability Corporation, "2018 Compliance Monitoring and Enforcement Program Annual Report," https://www.nerc.com/pa/comp/Resources/Pages/default.aspx, 2019, [Online; accessed July, 2019].

[25] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the one big switch abstraction in software-defined networks," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 2013, pp. 13–24.

[26] S. Shenker, M. Casado, T. Koponen, N. McKeown et al., "The future of networking, and the past of protocols," *Open Networking Summit*, vol. 20, pp. 1–30, 2011.

[27] R. Ahlswede, N. Cai, S.-Y. Li, and R. W. Yeung, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[28] R. Koetter and M. Médard, "An algebraic approach to network coding," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 5, pp. 782–795, 2003.

[29] F. F. Wu, K. Moslehi, and A. Bose, "Power system control centers: Past, present, and future," *Proceedings of the IEEE*, vol. 93, no. 11, pp. 1890–1908, 2005.

[30] I. Morsi and L. M. El-Din, “SCADA system for oil refinery control,” *Measurement*, vol. 47, pp. 5–13, 2014.

[31] S. Avlonitis, M. Pappas, K. Moutesidis, D. Avlonitis, K. Kouroumbas, and N. Vlachakis, “PC based SCADA system and additional safety measures for small desalination plants,” *Desalination*, vol. 165, pp. 165–176, 2004.

[32] A. Swales et al., *Open Modbus/TCP specification*, Schneider Electric Std., 1999.

[33] IEEE Power and Energy Society, “IEEE 1815-2012,” https://standards.ieee.org/findstds/standard/1815-2012.html, 2019, [Online; accessed November, 2019].

[34] Y. Liang and R. H. Campbell, “Understanding and simulating the IEC 61850 standard,” University of Illinois, Urbana-Champaign, Tech. Rep., 2008.

[35] R. Kinney, P. Crucitti, R. Albert, and V. Latora, “Modeling cascading failures in the North American power grid,” *The European Physical Journal B-Condensed Matter and Complex Systems*, vol. 46, no. 1, pp. 101–107, 2005.

[36] P. Crucitti, V. Latora, and M. Marchiori, “A topological analysis of the Italian electric power grid,” *Physica A: Statistical Mechanics and its Applications*, vol. 338, no. 1-2, pp. 92–97, 2004.

[37] T. G. Griffin and B. J. Premore, “An experimental analysis of BGP convergence time,” in *Proceedings Ninth International Conference on Network Protocols. ICNP 2001.* IEEE, 2001, pp. 53–61.

[38] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, “Delayed Internet routing convergence,” *IEEE/ACM Transactions on Networking*, vol. 9, no. 3, pp. 293–306, 2001.

[39] D. L. Tennenhouse and D. J. Wetherall, “Towards an active network architecture,” *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 2, pp. 5–17, 1996.

[40] T. Lakshman, T. Nandagopal, R. Ramjee, K. Sabnani, and T. Woo, “The SoftRouter architecture,” in *Proc. ACM SIGCOMM Workshop on Hot Topics in Networking*, vol. 2004. Citeseer, 2004.

[41] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe, “Design and implementation of a routing control platform,” in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2.* USENIX Association, 2005, pp. 15–28.

[42] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 5, pp. 41–54, 2005.

[43] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama et al., "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

[44] N. McKeown, T. Sloane, and J. Wanderer, "P4 runtime–putting the control plane in charge of the forwarding plane," 2017.

[45] "P4: Language Specification," https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2019, [Online; accessed November, 2019].

[46] "P4: Portable Switch Architecture Specification," https://p4.org/p4-spec/docs/PSA-v1.0.0.html, 2019, [Online; accessed November, 2019].

[47] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: static checking for networks," in *NSDI*, 2012, pp. 113–126.

[48] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[49] R. Kumar and D. M. Nicol, "Validating resiliency in software defined networks for smart grids," in *2016 IEEE International Conference on Smart Grid Communications*. IEEE, 2016, pp. 441–446.

[50] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A toolkit for firewall modeling and analysis," in *2006 IEEE Symposium on Security and Privacy*. IEEE, 2006, pp. 15–pp.

[51] H. Mai, "Diagnose network failures via data-plane analysis," M.S. thesis, University of Illinois, Urbana-Champaign, 2010.

[52] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and conquer to verify forwarding tables in huge networks," *NSDI*, 2014.

[53] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *NSDI*, 2013, pp. 99–111.

[54] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements," *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[55] A. Aydeger, K. Akkaya, M. H. Cintuglu, A. S. Uluagac, and O. Mohammed, "Software defined networking for resilient communications in Smart Grid active distribution networks," in *Communications (ICC), 2016 IEEE International Conference on.* IEEE, 2016, pp. 1–6.

[56] D. Gyllstrom, N. Braga, and J. Kurose, "Recovery from link failures in a Smart Grid communication network using OpenFlow," in *2014 IEEE International Conference on Smart Grid Communications.* IEEE, 2014, pp. 254–259.

[57] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks.* ACM, 2010, p. 19.

[58] "Open vSwitch," http://www.openvswitch.org.

[59] "Ryu SDN Framework," http://osrg.github.io/ryu/.

[60] C. Clos, "A study of non-blocking switching networks," *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[61] T. Elhourani, A. Gopalan, and S. Ramasubramanian, "IP fast rerouting for multi-link failures," in *INFOCOM, 2014 Proceedings IEEE.* IEEE, 2014, pp. 2148–2156.

[62] IEEE Power and Energy Society, "IEEE C37.118," https://standards.ieee.org/findstds/standard/C37.118.1a-2014.html, 2019, [Online; accessed November, 2019].

[63] C. K. Veitch, J. M. Henry, B. T. Richardson, and D. H. Hart, "Microgrid cyber security reference architecture," Sandia National Laboratories, Tech. Rep. SAND2013-5472, 2013.

[64] The ONOS project, "ONOS-VPLS," https://wiki.onosproject.org/display/ONOS/VPLS+User+Guide, 2019, [Online; accessed November, 2019].

[65] H. Cancela, M. E. Khadiri, G. Rubino, and B. Tuffin, "Balanced and approximate zero-variance recursive estimators for the network reliability problem," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 25, no. 1, p. 5, 2015.

[66] D. M. Nicol and R. Kumar, "Efficient Monte Carlo evaluation of sdn resiliency," in *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation.* ACM, 2016, pp. 143–152.

[67] D. M. Nicol and R. Kumar, “SDN resiliency to controller failure in mobile contexts,” in *Proceedings of the 2019 Winter Simulation Conference*. IEEE Press, 2019.

[68] M. O. Ball, “Computational complexity of network reliability analysis: An overview,” *IEEE Transactions on Reliability*, vol. 35, no. 3, pp. 230–239, 1986.

[69] S. Asmussen and P. W. Glynn, *Stochastic Simulation: Algorithms and Analysis*. Springer Science & Business Media, 2007.

[70] T. Qian, F. Mueller, and Y. Xin, “A Linux real-time packet scheduler for reliable static SDN routing,” in *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, 2017, pp. 25:1–25:22.

[71] D. Jin and D. M. Nicol, “Parallel simulation of software defined networks,” in *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 2013, pp. 91–102.

[72] “gRPC: a high-performance, open-source, universal RPC framework,” https://grpc.io/.

[73] C. H. Hauser, D. E. Bakken, and A. Bose, “A failure to communicate: next generation communication requirements, technologies, and architecture for the electric power grid,” *IEEE Power and Energy Magazine*, vol. 3, no. 2, pp. 47–55, 2005.

[74] ARINC, “Aircraft Data Network, Avionics Full Duplex Switched Ethernet (AFDX) Network,” 2003.

[75] I. Land and J. Elliott, “Architecting ARINC 664, part 7 (AFDX) solutions.” Xilinx, 2009.

[76] H. Charara, J. L. Scharbarg, J. Ermont, and C. Fraboul, “Methods for bounding end-to-end delays on an AFDX network,” in *18th Euromicro Conference on Real-Time Systems (ECRTS’06)*, 2006, pp. 10 pp.–202.

[77] N. Instruments, “Controller Area Network (CAN) Overview.” [Online]. Available: http://www.ni.com/white-paper/2732/en/

[78] “The netperf homepage,” 2019, https://hewlettpackard.github.io/netperf/, [Online; accessed November, 2019].

[79] J.-Y. Le Boudec and P. Thiran, *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer Science & Business Media, 2001, vol. 2050.

[80] S. Chen and K. Nahrstedt, "On finding multi-constrained paths," in *Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on*, vol. 2. IEEE, 1998, pp. 874–879.

[81] *Enhancements for Scheduled Traffic*, http://www.ieee802.org/1/pages/802.1bv.html, IEEE Std. 802.11Qbv, 2015, [Online; accessed November, 2019].

[82] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach," in *Global Communications Conference (GLOBECOM), 2013 IEEE.* IEEE, 2013, pp. 1397–1402.

[83] J. W. Guck and W. Kellerer, "Achieving end-to-end real-time quality of service with software defined networking," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on.* IEEE, 2014, pp. 70–76.

[84] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *NSDI*, 2015, pp. 1–14.

[85] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 435–448, 2015.

[86] "FlexRay Automotive Communication Bus Overview," http://www.ni.com/white-paper/3352/en/, [Online; accessed November, 2019].

[87] J. M. Jaffe, "Algorithms for finding paths with multiple constraints," *Networks*, vol. 14, no. 1, pp. 95–116, 1984.

[88] *Standard for Ethernet*, http://www.ieee802.org/3/, IEEE Std. 802.3, 2012, [Online; accessed November, 2019].

[89] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 7–12.

[90] R. L. S. De Oliveira, A. A. Shinoda, C. M. Schweitzer, and L. R. Prete, "Using Mininet for emulation and prototyping software-defined networks," in *IEEE Colombian Conference on Communications and Computing (COLCOM).* IEEE, 2014, pp. 1–6.

[91] M. Erel, E. Teoman, Y. Özçevik, G. Seçinti, and B. Canberk, "Scalability analysis and flow admission control in Mininet-based SDN environment," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on.* IEEE, 2015, pp. 18–19.

[92] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos, "SDDC: A software defined datacenter experimental framework," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on.* IEEE, 2015, pp. 189–194.

[93] E. Jo, D. Pan, J. Liu, and L. Butler, "A simulation and emulation study of SDN-based multipath routing for fat-tree data center networks," in *Proceedings of the 2014 Winter Simulation Conference.* IEEE Press, 2014, pp. 3072–3083.

[94] K. Leggett, R. Moxley, and D. Dolezilek, "Station device and network communications performance during system stress conditions," in *Proceedings of the Protection, Automation and Control World Conference, Dublin, Ireland*, 2010.

[95] T. Elhourani, A. Gopalan, and S. Ramasubramanian, "IP fast rerouting for multi-link failures," *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 3014–3025, 2016.

[96] Y.-A. Pignolet, S. Schmid, and G. Tredan, "Load-optimal local fast rerouting for resilient networks," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on.* IEEE, 2017, pp. 345–356.

[97] M. Chiesa, I. Nikolaevskiy, S. Mitrović, A. Panda, A. Gurtov, A. Maidry, M. Schapira, and S. Shenker, "The quest for resilient (static) forwarding tables," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on.* IEEE, 2016, pp. 1–9.

[98] M. Chiesa, A. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, M. Shapira, and S. Shenker, "On the resiliency of randomized routing against multiple edge failures," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 55. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[99] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, "XORs in the air: practical wireless network coding," *IEEE/ACM Transactions on Networking (ToN)*, vol. 16, no. 3, pp. 497–510, 2008.

[100] P. Ostovari, J. Wu, and A. Khreishah, "Network coding techniques for wireless and sensor networks," in *The Art of Wireless Sensor Networks.* Springer, 2014, pp. 129–162.

[101] J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros, "Network coding meets TCP," in *INFOCOM 2009, IEEE.* IEEE, 2009, pp. 280–288.

[102] P. A. Chou, Y. Wu, and K. Jain, "Practical network coding," in *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, vol. 41, no. 1. The University; 1998, 2003, pp. 40–49.

[103] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4FPGA: A rapid prototyping framework for p4," in *Proceedings of the Symposium on SDN Research.* ACM, 2017, pp. 122–135.

[104] Barefoot Networks, "Tofino," https://barefootnetworks.com/products, 2019, [Online; accessed November, 2019].

[105] M. Hadley, D. Nicol, and R. Smith, "Software-defined networking redefines performance for Ethernet control systems," in *Power and Energy Automation Conference*, 2017.

[106] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on.* IEEE, 2014, pp. 61–66.

[107] R. Kumar, M. Hasan, S. Padhy, K. Evchenko, L. Piramanayagam, S. Mohan, and R. B. Bobba, "End-to-end network delay guarantees for real-time systems using SDN," in *2017 IEEE Real-Time Systems Symposium (RTSS).* IEEE, 2017, pp. 231–242.

[108] J. Krigslund, J. Hansen, D. E. Lucani, F. H. Fitzek, and M. Médard, "Network coded software defined networking: Design and implementation," in *European Wireless 2015; 21th European Wireless Conference; Proceedings of.* VDE, 2015, pp. 1–6.

[109] C. Gkantsidis, J. Miller, and P. Rodriguez, "Comprehensive view of a live network coding P2P system," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement.* ACM, 2006, pp. 177–188.

[110] C. Gkantsidis and P. R. Rodriguez, "Network coding for large scale content distribution," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 4. IEEE, 2005, pp. 2235–2245.

[111] C. Wu, B. Li, and Z. Li, "Dynamic bandwidth auctions in multioverlay P2P streaming with network coding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 6, pp. 806–820, 2008.

[112] L. Zhang, S. Lai, C. Wu, Z. Li, and C. Guo, "Virtualized network coding functions on the internet," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 129–139.

[113] R. Prior, D. E. Lucani, Y. Phulpin, M. Nistor, and J. Barros, "Network coding protocols for smart grid communications," *IEEE Transactions on Smart Grid*, vol. 5, no. 3, pp. 1523–1531, 2014.

[114] Y. Phulpin, J. Barros, and D. Lucani, "Network coding in smart grids," in *2011 IEEE International Conference on Smart Grid Communications*. IEEE, 2011, pp. 49–54.

[115] H. Su and X. Zhang, "Network coding based QoS-provisioning MAC for wireless smart metering networks," in *International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness*. Springer, 2010, pp. 161–171.

[116] H. Giesen, L. Shi, J. Sonchack, A. Chelluri, N. Prabhu, N. Sultana, L. Kant, A. J. McAuley, A. Poylisher, A. DeHon et al., "In-network computing to the rescue of faulty links," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*. ACM, 2018, pp. 1–6.

[117] D. Zhuo, M. Ghobadi, R. Mahajan, A. Phanishayee, X. K. Zou, H. Guan, A. Krishnamurthy, and T. E. Anderson, "Rail: A case for redundant arrays of inexpensive links in data center networks." in *NSDI*, 2017, pp. 561–576.

[118] Z. Wang and J. Crowcroft, "Quality-of-service routing for supporting multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1228–1234, 1996.

[119] S.-Y. Li, R. W. Yeung, and N. Cai, "Linear network coding," *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371–381, 2003.

[120] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong, "A random linear network coding approach to multicast," *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

[121] E. Ayanoglu, I. Chih-Lin, R. D. Gitlin, and J. E. Mazo, "Diversity coding for transparent self-healing and fault-tolerant communication networks," *IEEE Transactions on Communications*, vol. 41, no. 11, pp. 1677–1686, 1993.

[122] "AquaFlow," https://github.com/gopchandani/AquaFlow, 2019, [Online; accessed November, 2019].

[123] "Behavioral Model Repository," https://github.com/p4lang/behavioral-model, 2019, [Online; accessed November, 2019].

[124] "Scapy: the Python-based interactive packet manipulation program and library," https://github.com/secdev/scapy, 2019, [Online; accessed November, 2019].

[125] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[126] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, 2016.

[127] S. Chinchali, P. Hu, T. Chu, M. Sharma, M. Bansal, R. Misra, M. Pavone, and S. Katti, "Cellular network traffic scheduling with deep reinforcement learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[128] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications Surveys & Tutorials*, 2019.

[129] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.